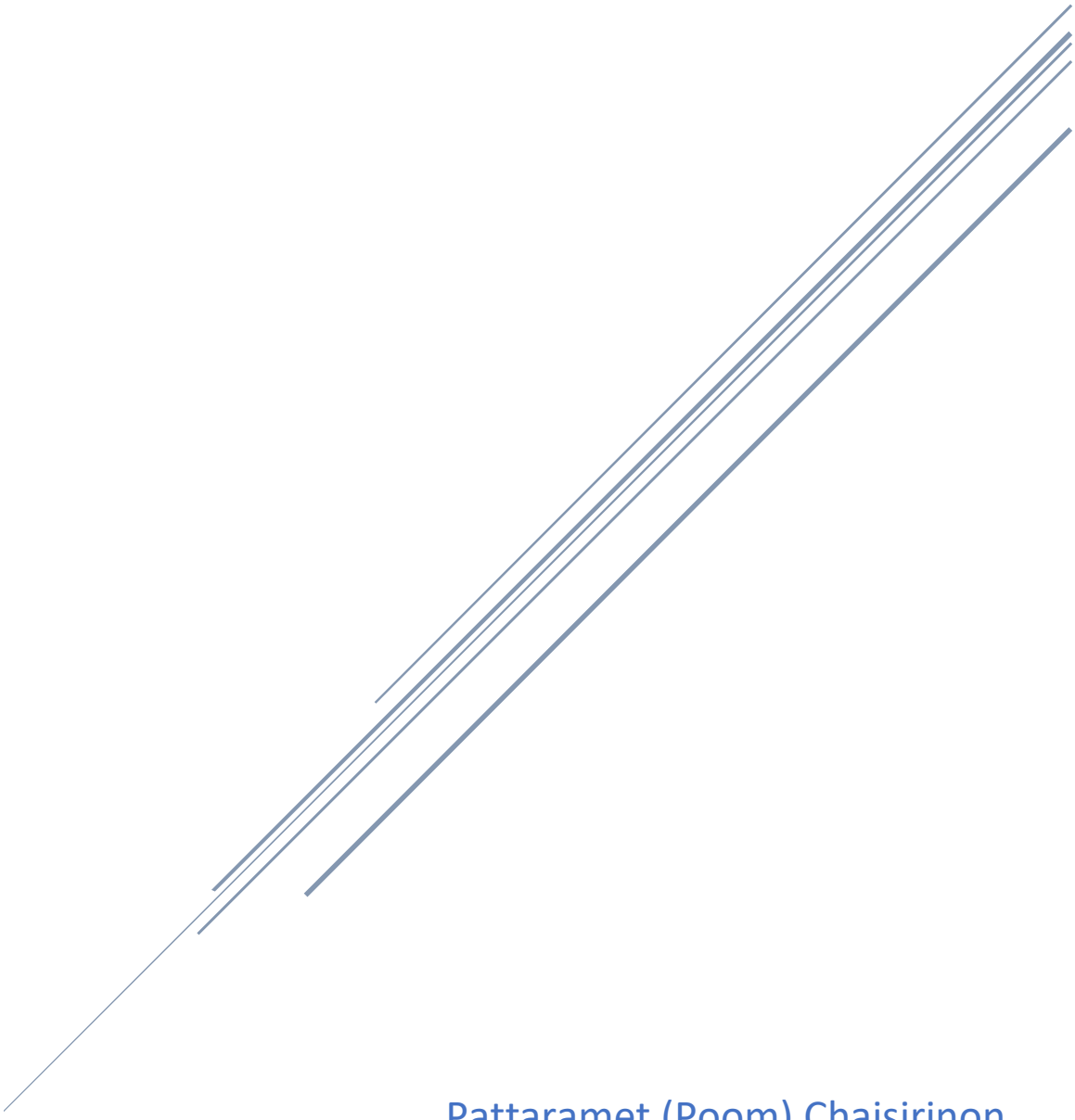
AMAZOOM AUTOMATED WAREHOUSE

CPEN 333 Design Document



Pattaramet (Poom) Chaisirinon
88528344

Contents

Executive Summary

Amazoom automated warehouse is a real-time simulation of a system that runs and controls an automated warehouse. For security reasons, the designed system needs authentication from the warehouse admin before the admin can initialize the warehouse set up. The system will take a file path and runs warehouse layout setup according to the given file if the file is valid. The file contains information on the basic layout structure of the warehouse, robots and trucks carrying capacities, and simulation specific instructions such as travel time and the speed up factor of the simulation for the purpose of this demo.

The warehouse central computer is the brain of the entire operation which syncs up with an online database powered by Google Cloud Firestore. There are two background threads that are only activated once there is a change in the database to listen for incoming orders and low stock alert. Additional two threads are used to regulate the loading dock and update asynchronously update order status to the database.

Modeled from real Amazon warehouses, the warehouse is designed to maximize efficiency and prioritized minimum wait time for customers to receive their orders. In order to do so, there will be equal numbers of robots working asynchronously as the numbers of columns within the warehouse. This eliminates unnecessary robot movements such as changing columns and increases storage capacity since extra pathways can be replaced by extra shelving units. Products' locations are randomly distributed as statistically, it will reduce travel distance and travel time. Most importantly, not all products in an order have to be loaded to the shipping truck before being shipped. This ensure that the customers will get their products as soon as possible in the unlikely event of system downtime in certain sectors. The warehouse can easily be scaled up by increasing the number of rows without having to increase the number of robots.

Once a new order is received, the central computer splits up the order by product locations and assign them to the appropriate robot in each sector. Each robot is designed to find the shortest path to each product regardless of the order the job is assigned to the robot. However, robots will always prioritize restocking jobs over retrieving jobs to ensure that inventory trucks leave to dock as quickly as possible for new shipping trucks to dock. Robots work completely asynchronously as one complete order is not assigned to only one robot. This minimizes robots' idle time as well as speeds up the product retrieval process. Once all products from an order is loaded to a designated shipping truck, the central computer updates the order's status on the database which will be reflected on both the customer and admin user interface.

The central computer listens for low stock alerts from the database and automatically order products according to the automatic restock setting – defaults to a full restock. Once the inventory trucks are loaded and docked, the central computer assigns restocking jobs to the appropriate robot to do a randomly distributed restock of products.

Finally, the customer and admin mobile user interface is available for iOS and android smartphones and tablets. The admin user interface captures and animates real-time robot position inside the warehouse. The admin user interface also allows for order query, stock query, automatic restock setting, and low stock alert. On the other hand, the customer user interface is a fully interactive user interface that supports multiple users with unique authentication information and allows for product browsing, adding or removing products to cart, automatically reserve products in cart, and placing orders.

Warehouse Setup File

```
5 8 6 30 1000 10 200
//==========//
Initialization: rows, columns, shelves, robotCapacity, travelTime, TruckVolume, TruckWeight
//==========//

Warehouse Layout Details:

      1     2    3     4     5     6     7     8
    ====  ====  ====  ====  ====  ====  ====  ====
1   |     ||    ||    ||    ||    ||    ||    ||    |
    |     ||    ||    ||    ||    ||    ||    ||    |
2   |     ||    ||    ||    ||    ||    ||    ||    |
    |     ||    ||    ||    ||    ||    ||    ||    |
3   |     ||    ||    ||    ||    ||    ||    ||    |
    |     ||    ||    ||    ||    ||    ||    ||    |
4   |     ||    ||    ||    ||    ||    ||    ||    |
    |     ||    ||    ||    ||    ||    ||    ||    |
5   |     ||    ||    ||    ||    ||    ||    ||    |
    |     ||    ||    ||    ||    ||    ||    ||    |
6   |****||****||****||****||****||****||****||****|
    |     ||    ||    ||    ||    ||    ||    ||    |
              ^^^^^^^^^^^^^^^^^^^^^^^^^^

==== : charging doc

|| : shelves

**** : virtual gate for collision avoidance

^^^^ : loading docks
```

Use Cases Diagram

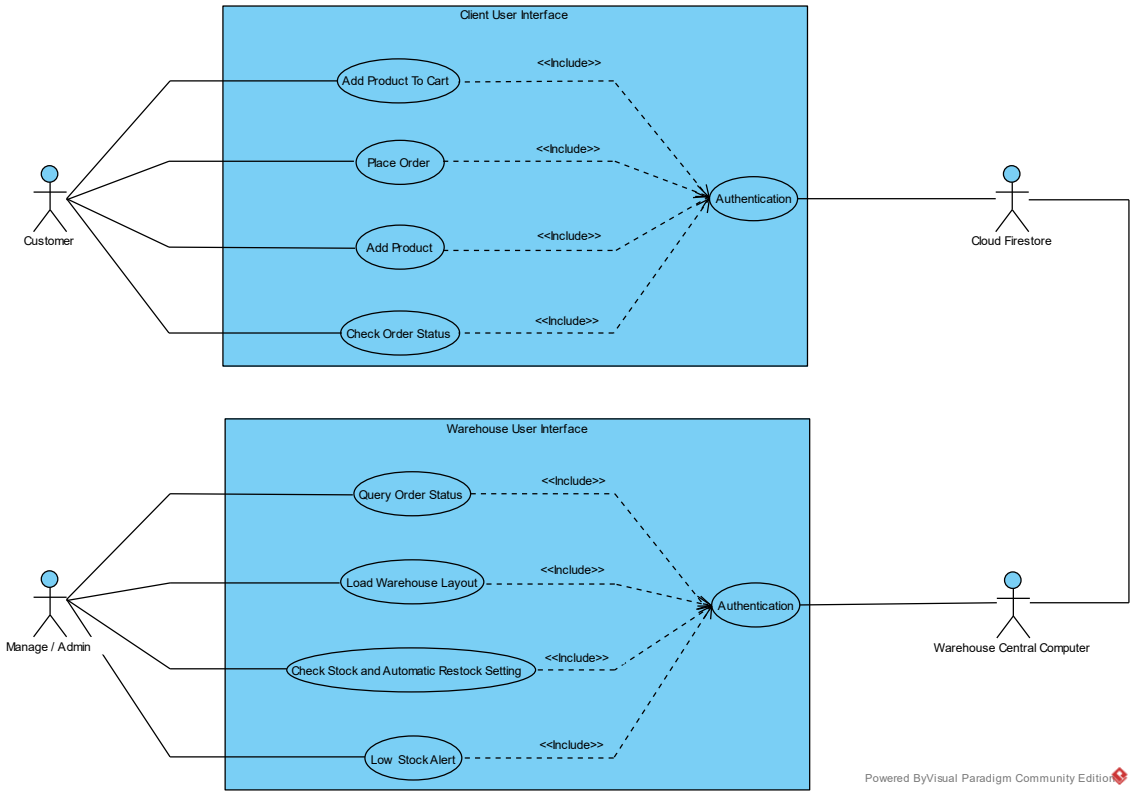


FIGURE 1: USE CASE DIAGRAM FOR USER AND ADMIN

Use Cases and Use Case Scenario Description

Identify User (Include)

1. The admin or user authenticates oneself using an email and password log in through the user interface. If already logged in, then **<<Scenario 1>>**
2. The system authenticates the admin through Cloud Firestore using the provided email and password
3. If the system cannot access the admin or user account from Cloud Firestore, then **<<Scenario 2>>**
4. If the admin authentication exists, then **<<Scenario 3>>**
5. If the user authentication exists, then **<<Scenario 4>>**
6. If Identification fails, then **<<Scenario 5>>**

End of Identify User

Scenario 1: Skip authentication steps are resume primary scenario

Scenario 2: Console print an error message. End-of-Scenario.

Scenario 3: Redirect admin to the admin dashboard. End-of-Scenario

Scenario 4: Redirect user to the user browse page. End-of-Scenario

Scenario 5: Console print and error message. End-of-Scenario

Use Cases: Warehouse User Interface

Load Warehouse Layout: (Use Case)

- The admin identifies oneself using a password into the Console terminal
- The admin provides a path containing the warehouse setup file
- The system attempts to read, validate, and interpret the file from the file path
- The system informs the admin about the problem and prompts the admin for a new file path if an error occurs with the file
- If acceptable, the system runs its start up sequence and set up the warehouse, fetch products, initialize robots, and start up trucks

Start: Load Warehouse Layout: (Use Case Scenario)

1) The admin authenticates oneself to gain access into the system
2) The admin loads a file containing warehouse information into the system by providing a file path
3) The system attempts to read, validate, and interpret the file from the given file path
4) If the system cannot read the file, then **<<Scenario 1>>**
5) The system inform admin with a success message

End of Load Warehouse Layout

Scenario 1: The system prints out an error message with information regarding the error and prompts the admin to enter a new file path

Query Order Status: (Use Case)

- The admin must have already authenticated oneself before the system grants access to this feature when selected
- The system performs a query to Cloud Firestore to fetch all user orders from the "User Orders" collection
- All user orders and their corresponding shipping status will be presented to the admin through a user interface

Start: Query Order Status: (Use Case Scenario)

1. **Include** **Identify User** (a prerequisite or precondition for the execution of this use case)
2. The admin can select the "Query Order Status" option from the user interface
3. The system fetch order ids and their corresponding shipping status from Cloud Firestore
4. If the system fails to query orders from database, then **<<Scenario 1>>**
5. The system displays the order status to the admin

End of Query Order Status

Scenario 1: Console print the error message. End of Query Order Status.

Check Stock and Automatic Restock Setting: (Use Case)

- The admin must have already authenticated oneself before the system grants access to this feature when selected
- The system performs a query to Cloud Firestore to fetch all product information from the "All products" collection
- All products and their corresponding stock count will be displayed
- The admin can interact with the automatic restock setting by tapping the product box once to increment automatic restocking count by +1 or double tap to decrement the automatic restocking count by -1

Start: Check Stock and Automatic Restock Setting: (Use Case Scenario)

1. **Include** **Identify User** (a prerequisite or precondition for the execution of this use case)
2. The admin can select the "Check Stock" option from the user interface
3. The system fetches all products and their corresponding remaining stock from Cloud Firestore
4. If the system fails to query products and stocks from database, then **<<Scenario 1>>**
5. The system displays the order status to the admin
6. The admin can increment, or decrement automatic restock setting by tapping to increment by +1 and double tapping to decrement by -1
7. The system updates the setting on to Cloud Firestore
8. If the system fails to update admin's setting to Cloud Firestore, then **<<Scenario 2>>**

End of Check Stock and Automatic Restock Setting

Scenario 1: Console print the error message. End of Check Stock and Automatic Restock Setting.

Scenario 2: Console print the error message. End of Check Stock and Automatic Restock Setting.

Low Stock Alert: (Use Case)

- The admin must have already authenticated oneself before the system grants access to this feature when selected
- The system performs a query to Cloud Firestore to fetch all product information from the "All products" collection
- Products that has a stock of less than or equal to the user setting at initial warehouse set up will be displayed

Start: Low Stock Alert: (Use Case Scenario)

1. **Include** **Identify User** (a prerequisite or precondition for the execution of this use case)
2. The system will query for products that is considered "low stock"
3. If the system fails to query products and stocks from database, then **<<Scenario 1>>**
4. The system displays the low stock to the admin in a dialogue box on the user interface

End of Low Stock Alert

Scenario 1: Console print the error message. End of Low Stock Alert.

Check Robot Positions: (Use Case)

- The admin must have already authenticated oneself before the system grants access to this feature
- The system listens to updates of each robot's location from Cloud Firestore in "All Robots" collection and display them on to a grid display on the user interface
- The admin will be able to observe each robot's location in the warehouse

Start: Check Robot Position: (Use Case Scenario)

1. **Include** **Identify User** (a prerequisite or precondition for the execution of this use case)
2. The system will query for all robot positions and continuously listen to updates in the background
3. If the system fails to query products and stocks from database, then **<<Scenario 1>>**
4. The system updates the display if there is a change in robot's position and repeat until admin logs out

End of Check Robot Position

Scenario 1: Console print the error message. End of Low Stock Alert.

Use Case: Client User Interface

Add Product to Cart: (Use Case)

- The user must have already authenticated oneself before the system grants access to this feature
- The system displays all available products in stock on to the user interface
- The user can tap once to add the product to cart

- The system will decrement and update the stock on Cloud Firestore accordingly as the user adds products to the cart to signify that the products have been reserved by a specific user when added to cart
- The user can remove item from cart by double tapping the item
- The system will update the stock on Cloud Firestore accordingly

Start: Add Product to Cart: (Use Case Scenario)

1. **Include** **Identify User** (a prerequisite or precondition for the execution of this use case)
2. The system will query all products from "All products" collection on Cloud Firestore and display products onto the user interface
3. If the product's stock is zero, then **<<Scenario 1>>**
4. The user can tap to add product to cart or double tap to remove the product from the cart **<<Scenario 2>>**
5. The system will instantly update product's stock accordingly in Cloud Firestore

End of Add Product to Cart

Scenario 1: Product will not be displayed

Scenario 2: The system will update the numbers of items in cart and calculate the total price

Place Order: (Use Case)

- The user must have already authenticated oneself before the system grants access to this feature and there must already be items added to cart by the user
- The user confirms and place an order and click the "Place Order" button
- The system will update Cloud Firestore and adds a new order to the "User Orders" collection

Start: Place Order: (Use Case Scenario)

1. **Include** **Identify User** (a prerequisite or precondition for the execution of this use case)
2. **Include Add Product to Cart** (a prerequisite or precondition for the execution of this use case)
3. The user can press the "Place Order" button to place order
4. The system will relay the information to Cloud Firestore and notify the warehouse of a new incoming order
5. If the system fails to update order to Cloud Firestore, then **<<Scenario 1>>**

End of Place Order

Scenario 1: Console print the error message. End of Place Order

Check Order Status: (Use Case)

- The user must have already authenticated oneself before the system grants access to this feature
- The user selects "Check Order Status" option
- The system displays all orders associated with the currently logged in unique user id and their shipping status as true for shipped and false for pending

Start: Check Order Status: (Use Case Scenario)

1. **Include** Identify User (a prerequisite or precondition for the execution of this use case)
2. The user selects the “Check Order Status” icon on the user interface
3. The system will query for all the orders that belong to the currently signed in user id on Cloud Firestore under the “User Orders” collection
4. If the system fails to query orders and order status from Cloud Firestore, then **<<Scenario 1>>**
5. The system displays all user orders and their corresponding shipment status for the currently signed in user

End of Check Order Status

Scenario 1: Console print the error message. End of Check Order Status

Add Product: (Use Case)

- The user must have already authenticated oneself before the system grants access to this feature
- The user selects the “Add” icon to add default items (for the purpose of the demo) to Cloud Firestore
- The system will add the product to Cloud Firestore
- The system will update and display the newly added product on the user interface

Start: Add Product: (Use Case Scenario)

1. **Include** Identify User (a prerequisite or precondition for the execution of this use case)
2. The user selects the “Add” icon button to add default items
3. The system adds the default item to Cloud Firestore
4. If the system fails to query orders and order status from Cloud Firestore, then **<<Scenario 1>>**
5. The system updates its current list of available products and display an updated version of the list on the user interface
6. If the system fails to query orders and order status from Cloud Firestore, then **<<Scenario 2>>**

End of Add Product

Scenario 1: Console print the error message. End of Add Product

Scenario 2: Console print the error message. End of Add Product

Class Diagram

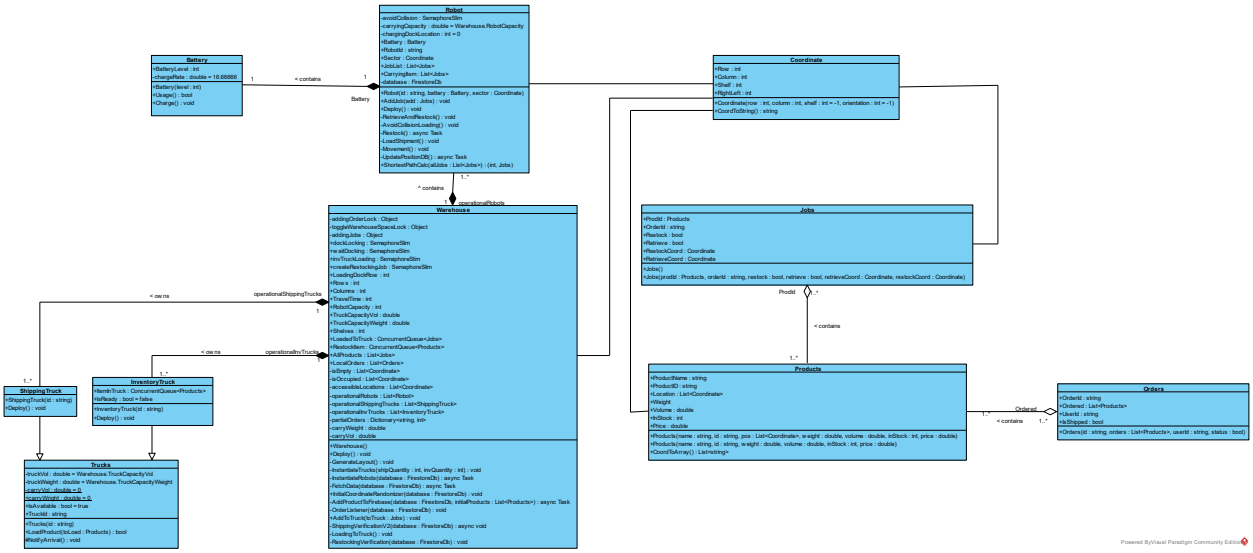


FIGURE 2: CLASS DIAGRAM

Object Interaction Diagram

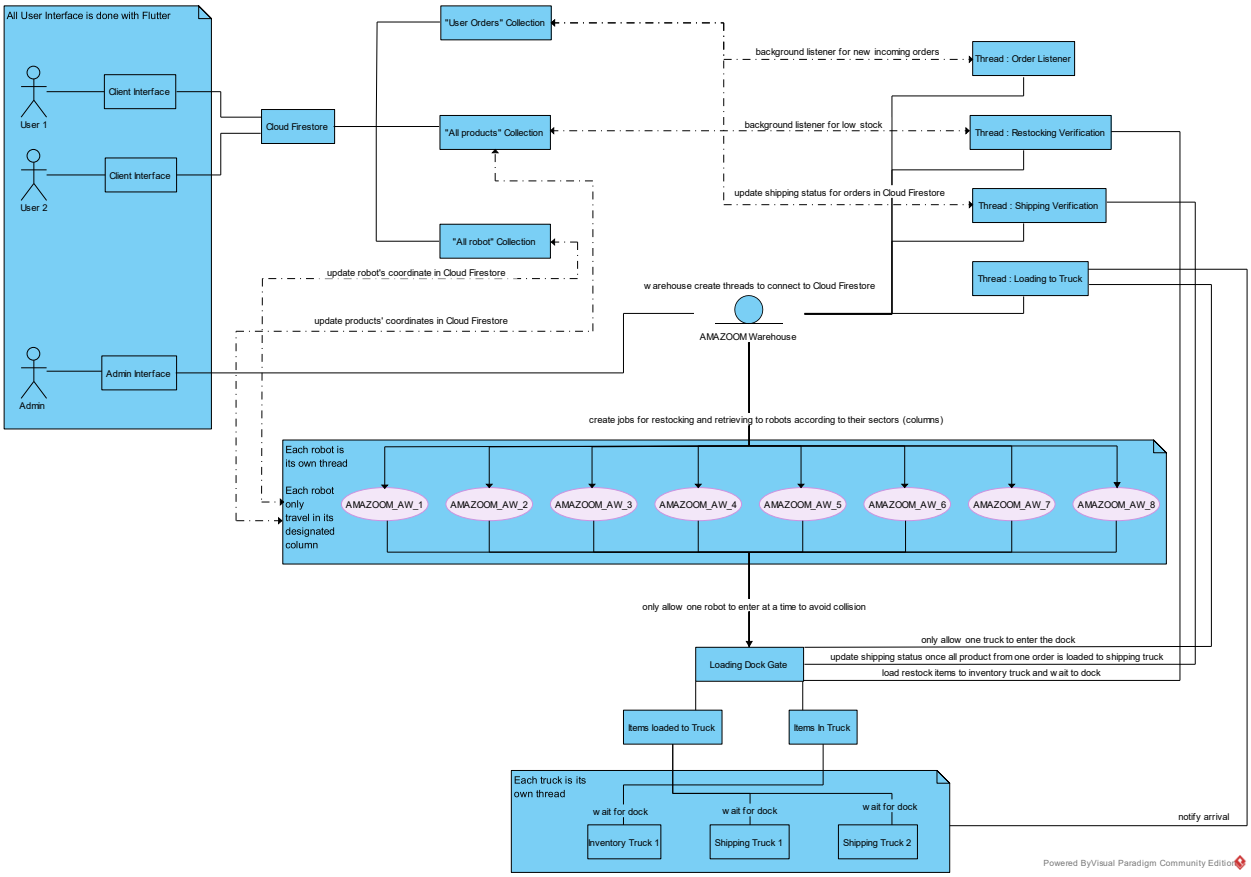


FIGURE 3: OBJECT INTERACTION DIAGRAM

Admin User Interface Design

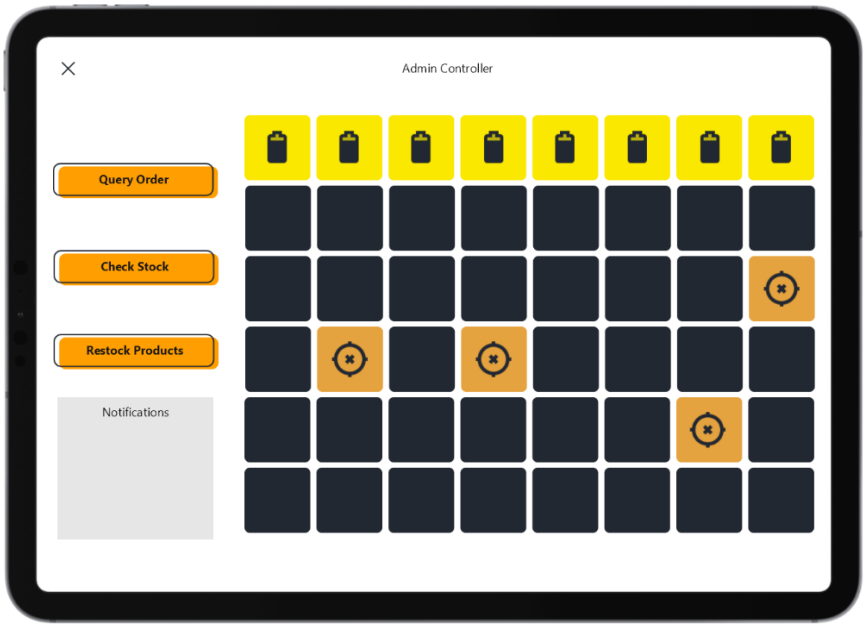


FIGURE 4: ADMIN CONTROLLER USER INTERFACE DESIGN

Client User Interface Design

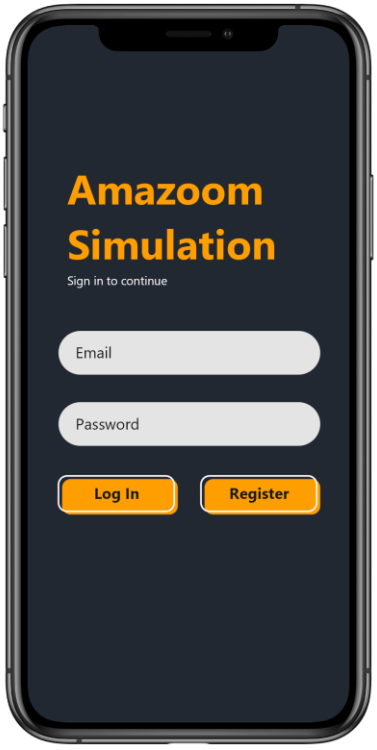


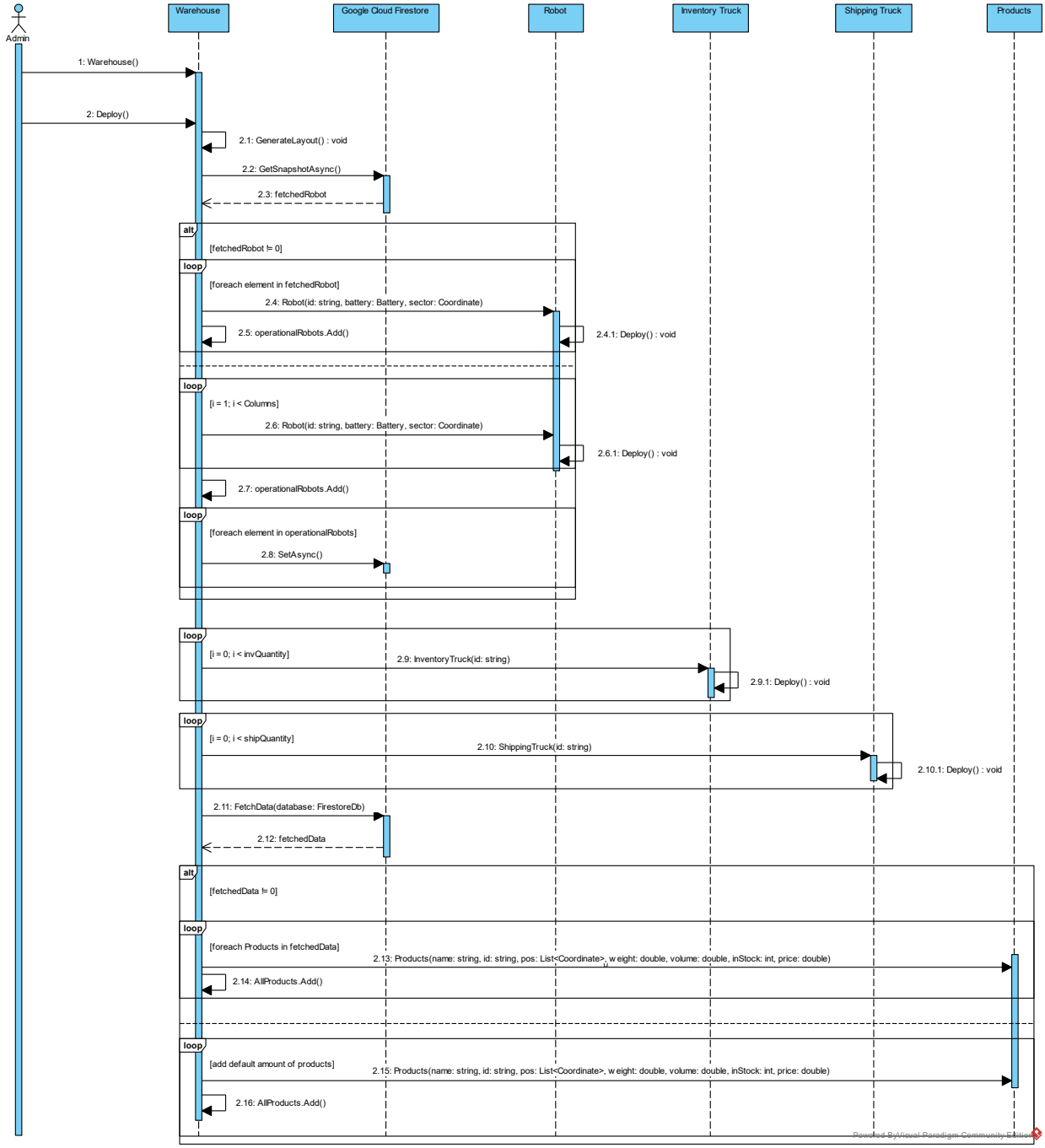
FIGURE 5: LOG IN SCREEN

FIGURE 6: BROWSE SCREEN

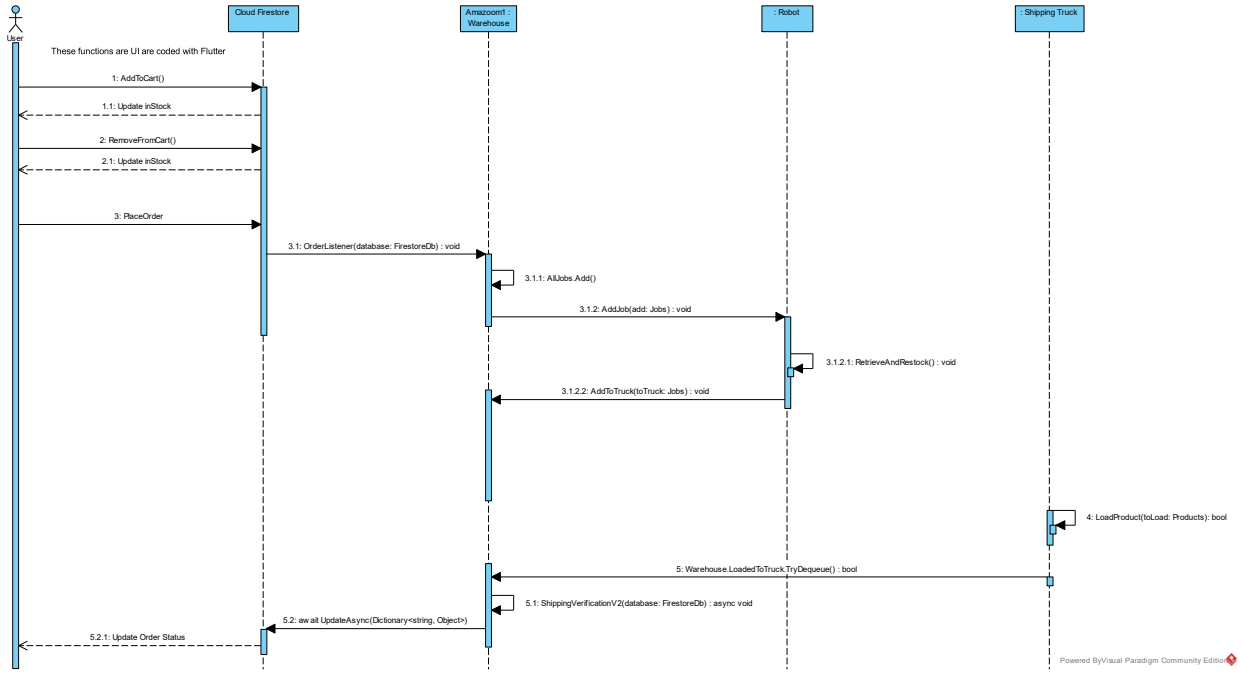
FIGURE 7: CART SCREEN

Sequence Diagram

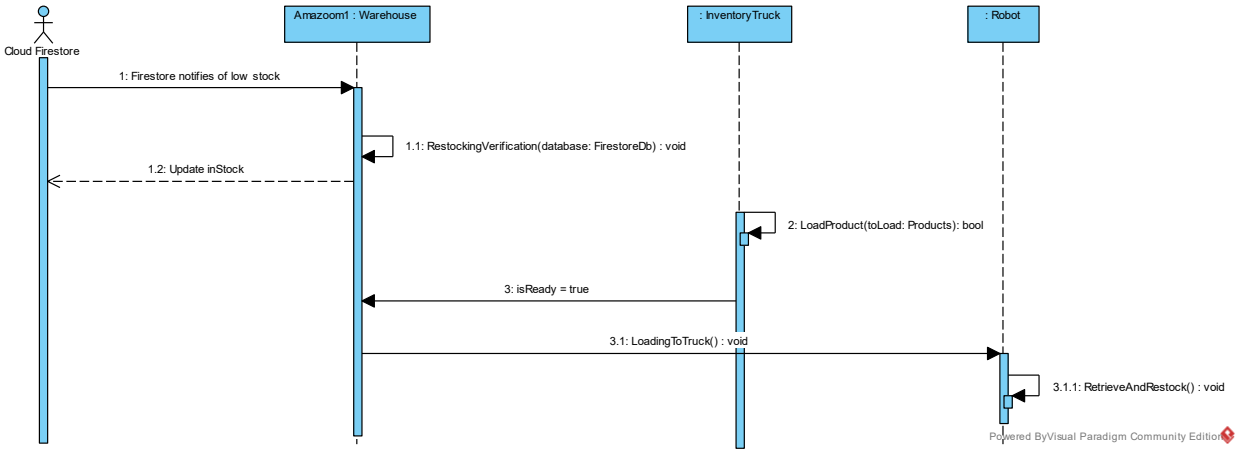
Warehouse Setup



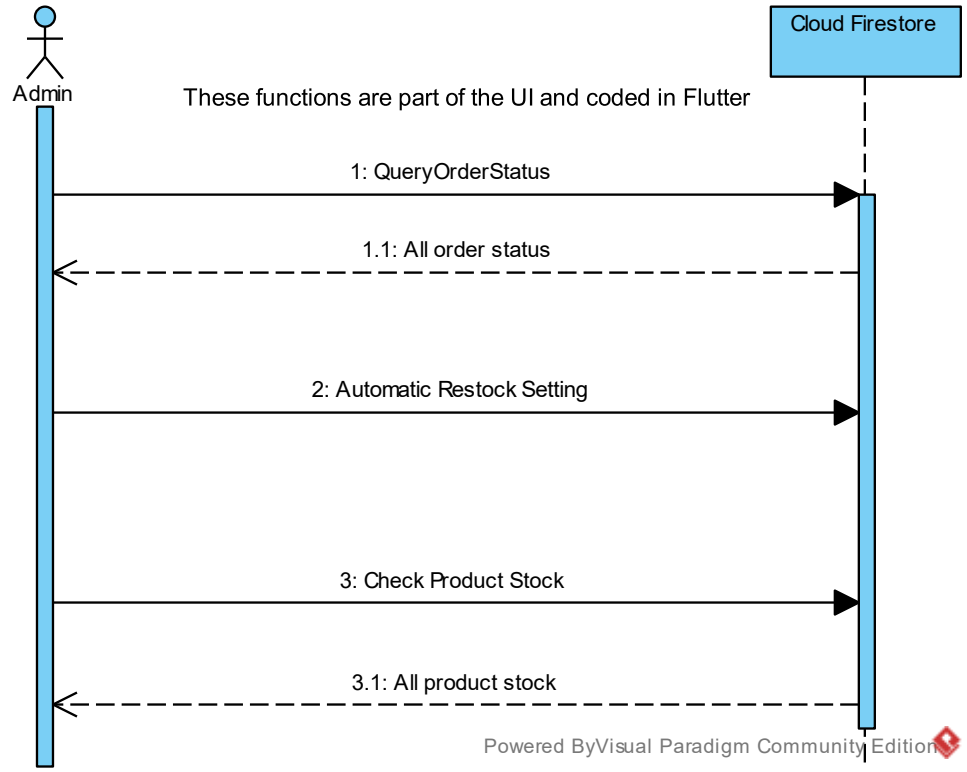
Ordering and Retrieving Orders in Warehouse



Restocking



Admin Query



Method Specifications

All methods specifications for the automated warehouse system are included in detail in the source code available in Github or in the Source Code section of this documentation.

Formal Testing

*Note: Coordinates are given in the following format [Row, Column, Shelf, Orientation]

Cloud Firestore Connection

Establishing a connection to Cloud Firestore

- Used a try/catch block to handle connection errors in case connection is not successfully established

Robot Class

ShortestPathCalc()

- Testing restock priority; Expected that robot should execute restocks before retrieve
    - Gave the robot a List<Jobs> { restock1, retrieve1, retrieve2, restock2, retrieve3 }
    - The algorithm did prioritize restocks job first as the robot executes restock1, restock2, and restock3 in this order then shortest path retrieval for retrieve1 and retrieve2
- Testing for shortest path retrieval

- Gave the robot a List<Jobs> { retrieve1, retrieve2, retrieve3}
  - Retrieve1: [4, 2, 6, 1]
  - Retrieve2: [5, 2, 1, 2]
  - Retrieve3: [1, 2, 4, 1]
  - Current Robot Location: [2, 2, -1, -1]
- Expected: The robot executes and retrieved retrieve3, retrieve1, retrieve2 in the following order
- Result: PASSED

Movement()

- Testing for correct output
  - Gave the robot a coordinate to travel to and the robot's current position
    - Destination: [1, 2, 5, 1]
    - Current Robot Location: [4, 2, 1, 2]
    - Destination: [4, 2, 1, 2]
    - Current Robot Location: [1, 2, 5, 1]
  - Expected: Console print "Position: 3, Robot id: 2" ; "Position: 2, Robot id: 2" ; "Position: 1, Robot id: 2" and "Position: 2, Robot id: 2" ; "Position: 3, Robot id: 2" ; "Position: 4, Robot id: 2"
  - Result: PASSED

Warehouse Class

InitialCoordinateRandomizer(FirestoreDb database)

- Testing for correct randomization
  - Having the system run and store unique randomized integers between 0 and 479 in a list
  - Expected: No duplicates in the list
  - Result: PASSED

AddProductToFirestore(FirestoreDb database, List<Products> initialProducts)

- Testing for syncing and updating data on to the online database
  - Gave a Dictionary<string, Object> to be updated to Cloud Firestore
  - Expected: Result shows up on Firestore and Console print a success message
  - Result: PASSED

FetchData(FirestoreDb database)

- Testing for database query and creating Coordinate class objects
  - Manually input testing data onto Firestore in a List/Array of strings format for storing coordinate class
  - Expected: A List<Coordinate> from List<Object> fetched from Firestore
  - Result: PASSED

OrderListener(FirestoreDb database)

- Testing for active listening function for updates on Cloud Firestore
    - Manually update testing data onto Firestore and Console print them as the updated testing data get fetched
    - Result: PASSED
- Testing for job creation and assigning them to the corresponding robots
    - Gave a valid Coordinate object
        - [1, 8, 2 ,2]
        - [5, 2, 1, 1]
    - Expected: Job assigned to robot 8 and robot 2 respectively
    - Result: PASSED

RestockVerification(FirestoreDb database)

- Testing for query of specific conditions from Cloud Firestore
    - Manually add 2 testing data onto Cloud Firestore and query only “inStock” less than or equal to 60
        - {“inStock”: 90}
        - {“inStock”: 60}
        - {“inStock”: 45}
    - Expected: Only fetched “inStock” with 60 and 45 and console print them
    - Result: PASSED

LoadingToTruck()

- Testing if only one truck can enter the docking area at a time
    - Created 3 tasks representing 3 trucks and running them
    - Expected: Only one truck console print “Truck id: {TruckId} entered docking” and “Truck id: {TruckId} left docking” at a time and no other truck prints “Truck id: {TruckId} entered docking” if the previous truck has not left. No deadlock should occur.
    - Result: PASSED

ShippingVerificationV2(FirestoreDb database)

- Testing for updating order status to Cloud Firestore
    - Expected: Order status “isShipped” only updates to Cloud Firestore when the robot loads all the products within that order into the shipping truck(s)
    - Result: PASSED

Source Code

Program

```
/// <summary>
/// You need to install Google.Cloud.Firestore NuGet Packages. Go to Project >> Manage NuGet Packages >> search for Google.Cloud.Firestore
/// Main method for instantiating the Warehouse object and deploying the warehouse
/// </summary>
namespace AmazoomDebug
{
    class Program
    {
        static void Main()
        {
            // initializes the warehouse
            Warehouse Amazoom1 = new Warehouse();
            Amazoom1.Deploy();
        }
    }
}
```

Battery

```
using System;
using System.Threading;

namespace AmazoomDebug
{
    /// <summary>
    /// unit = seconds
    /// max usage of 60 seconds from full charge (for the purpose of this demo, it is kept to be at 60 seconds)
    /// </summary>
    class Battery
    {
        // unit = seconds
        public int BatteryLevel { get; set; }
        private readonly double chargeRate = 16.66666;
        //private double drainRate = -1.666666;

        public Battery(int level)
        {
            BatteryLevel = level;
        }

        /// <summary>
        /// reporting battery level in 10% intervals; critical battery level is 10% and must be recharged
        /// max usage 60 seconds from full charge; equivalent to -10% every 6 steps the robot takes
        /// </summary>
        /// <returns>returns true if the battery is over 10% meaning the robot can be used. Otherwise return false</returns>
        public bool Usage()
        {
            if (BatteryLevel == 10)
            {
```

```
                return false;    // Doesn't allow usage once battery is 10%
            }
            else
            {
                BatteryLevel -= 10;
                Console.WriteLine("Battery: " + BatteryLevel);
            }
            return true;
        }

        /// <summary>
        /// Charging the battery from the current battery level to full battery.
        /// Charging time varies according to the current remaining battery level.
        /// Charging rate is 16.666 meaning it takes 6 seconds to fully charge the
battery
        /// </summary>
        public void Charge()
        {
            double chargeTime = (100 - BatteryLevel)/(chargeRate);
            Thread.Sleep(Convert.ToInt32(chargeTime));
            BatteryLevel = 100;
        }
    }
}
```

Coordinate

```
using System;

namespace AmazoomDebug
{
    /// <summary>
    /// Coordinate class for keeping track of locations in the warehouse
    /// </summary>
    public class Coordinate : IEquatable<Coordinate>
    {
        public int Row { get; set; }
        public int Column { get; set; }
        public int Shelf { get; set; }
        public int RightLeft { get; set; }    // right = 1 and left = 2

        public Coordinate(int row, int column, int shelf = -1, int orientation = -1)
        {
            Row = row;
            Column = column;
            Shelf = shelf;
            RightLeft = orientation;
        }

        /// <summary>
        /// Converting coordinate class into string
        /// </summary>
        /// <returns>String formatted as "row column shelf rightleft"</returns>
        public string CoordToString()
        {
            string coordniateString = Row + " " + Column + " " + Shelf + " " + RightLeft;

            return coordniateString;
```

```
        }

        /// <summary>
        /// Equals method override to check if the given object is equal to the current
object of Coordinate class
        /// </summary>
        /// <param name="obj"> object to be check with the current object of Coordinate
class</param>
        /// <returns>Return true if the two are equal as in same row, same column, and
same shelf interger values. Return false otherwise.</returns>
        public override bool Equals(object compareTo)
        {
            if (compareTo == null)
            {
                return false;
            }

            Coordinate convert = (Coordinate)compareTo;
            if (convert == null)
            {
                return false;
            }
            else
            {
                return Equals(convert);
            }
        }

        /// <summary>
        /// Get Hash Code method for Coordinate class object identification
        /// </summary>
        /// <returns>unique hash code for each coordinate class object</returns>
        public override int GetHashCode()
        {
            return Row + Column + Shelf + RightLeft;
        }

        /// <summary>
        /// Check if the given coordinate class matches with the current coordinate class
        /// </summary>
        /// <param name="other">Coordinate class object that will be compared to the
current Coordinate class object</param>
        /// <returns>Returns true if the two coordinates are the same. Otherwise returns
false</returns>
        public bool Equals(Coordinate other)
        {
            if (other == null)
            {
                return false;
            }

            return (this.Row.Equals(other.Row) && this.Column.Equals(other.Column) &&
this.Shelf.Equals(other.Shelf) && this.RightLeft.Equals(other.RightLeft));
        }
    }
}
```

Jobs

```
namespace AmazoomDebug
{
    /// <summary>
    /// Jobs containing information about restocking position, retreving position,
orderId, and product information
    /// A job is assigned to individual robots by the warehouse central computer
    /// </summary>
    class Jobs
    {
        public Products ProdId { get; set; }
        public string OrderId { get; set; }
        public bool Restock { get; set; }
        public bool Retrieve { get; set; }
        public Coordinate RestockCoord { get; set; }
        public Coordinate RetrieveCoord { get; set; }

        public Jobs()
        {
        }

        public Jobs (Products prodId, string orderId, bool restock, bool retrieve,
Coordinate retrieveCoord, Coordinate restockCoord)
        {
            ProdId = prodId;
            OrderId = orderId;
            Restock = restock;
            Retrieve = retrieve;
            RestockCoord = restockCoord;
            RetrieveCoord = retrieveCoord;
        }
    }
}
```

Orders

```
using System.Collections.Generic;

namespace AmazoomDebug
{
    /// <summary>
    /// Orders contain the orderId uniquely generated from Cloud Firestore, a list of
Products Class containing all the products in that order,
    /// the user id that corresponds to the order, and the shipping status that toggles
to true when all products in the order has been shipped
    /// </summary>
    class Orders
    {
        public string OrderId { get; set; }
        public List<Products> Ordered { get; set; } = new List<Products>();
        public string UserId { get; set; }
        public bool IsShipped { get; set; }

        public Orders(string id, List<Products> orders, string userId, bool status)
        {
            OrderId = id;
            Ordered = orders;
            UserId = userId;
```

```
            IsShipped = status;
        }
    }
}
```

Products

```
using System.Collections.Generic;

namespace AmazoomDebug
{
    /// <summary>
    /// Products Class contains information of each individual products
    /// It contains the product's name, id, a list of location within the warehose,
weight, volume, stock, and price
    /// </summary>
    class Products
    {
        public string ProductName { get; set; }
        public string ProductID { get; set; }
        public List<Coordinate> Location { get; set; } = new List<Coordinate>();
        public double Weight { get; set; }
        public double Volume { get; set; }
        public int InStock { get; set; }
        public double Price { get; set; }

        public Products(string name, string id, List<Coordinate> pos, double weight,
double volume, int inStock, double price)
        {
            ProductName = name;
            ProductID = id;
            Location = pos;
            Weight = weight;
            Volume = volume;
            InStock = inStock;
            Price = price;
        }

        public Products(string name, string id, double weight, double volume, int
inStock, double price)
        {
            ProductName = name;
            ProductID = id;
            Weight = weight;
            Volume = volume;
            InStock = inStock;
            Price = price;
        }

        /// <summary>
        /// Convert the list of coordinates containing all the locations of this specific
product inside the warehouse into a list of string
        /// </summary>
        /// <returns>List of string of coordinates formatted as "row column shelf
rightleft"</returns>
        public List<string> CoordToArray()
        {
            List<string> toArray = new List<string>();
```

```
            foreach (var coord in Location)
            {
                string temp = coord.Row + " " + coord.Column + " " + coord.Shelf + " " +
coord.RightLeft;

                toArray.Add(temp);
            }

            return toArray;
        }
    }
}
```

Robot

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using Google.Cloud.Firestore;

namespace AmazoomDebug
{
    /// <summary>
    /// Robot Class containing specific robot information such as its battery level,
sector, a List of current jobs, a list of items currently carrying,
    /// and the robot's capacity.
    ///
    /// This class automatically updates the robot's location to Cloud Firestore
    ///
    /// each moves takes 0.5 seconds for the purpose of the simulation; can be changed in
the Warehouse setup file
    /// </summary>
    class Robot
    {
        private static SemaphoreSlim avoidCollision = new SemaphoreSlim(1);
        private double carryingCapacity = Warehouse.RobotCapacity;    // max carrying
weight of 5kg; limited only by weight and not volume
        private int chargingDockLocation = 0;
        public Battery Battery { get; set; }
        public string RobotId { get; set; }
        public Coordinate Sector { get; set; }
        public List<Jobs> JobList { get; set; } = new List<Jobs>();
        public List<Jobs> CarryingItem { get; set; } = new List<Jobs>();
        private FirestoreDb database;

        public Robot(string id, Battery battery, Coordinate sector)
        {
            RobotId = id;
            Battery = battery;
            Sector = sector;

            // Established a connection to Cloud Firestore with a unique application id
            string path = AppDomain.CurrentDomain.BaseDirectory + @"amazoom-c1397-
firebase-adminsdk-ho7z7-6572726fc6.json";
            Environment.SetEnvironmentVariable("GOOGLE_APPLICATION_CREDENTIALS", path);
            database = FirestoreDb.Create("amazoom-c1397");
```

```
        }

        /// <summary>
        /// Adding jobs to the current list of jobs the robot has to finish
        /// </summary>
        /// <param name="add">Job class object to be added to the list</param>
        public void AddJob(Jobs add)
        {
            Console.WriteLine("Robot: " + Sector.Column + " received new job.");
            JobList.Add(add);
        }

        public void Deploy()
        {
            while (true)
            {
                if (JobList.Count != 0)
                {
                    RetrieveAndRestock();
                }
                else
                {
                    Movement(chargingDockLocation);
                    Battery.Charge();

                    Thread.Sleep(5000);    // robots become idle to release resources
from multi threading
                }
            }
        }

        /// <summary>
        /// Retrieve and restock jobs from specific location within the robot's column.
        /// Retrieve closest order first, if carrying capacity is full, then load the
shipping trucks.
        /// Always check distance between next destination and the loading docks, always
choose the shortest path if the robot is carrying a product
        /// </summary>
        private void RetrieveAndRestock()
        {
            // Item1: closestPath; Item2: corresponding Job
            var path = ShortestPathCalc(JobList);
            Jobs currentJob = path.Item2;

            // check distance from current location to the loading doc if the robot is
carrying something
            // else go pick up next item
            int toLoadingDock = Math.Abs(Sector.Row - Warehouse.LoadingDockRow);

            // if closestPath is to restock from the loading dock, then restock
            if (path.Item2.Restock)
            {
                Restock(currentJob).Wait();
            }
            // if closestPath is to retrive
            else
            {
```

```
                // Check if the closest path is to the loading dock or not and if the
robot is carrying an item
                // If so, load the current item to the truck first
                if (carryingCapacity < Warehouse.RobotCapacity && toLoadingDock <
path.Item1)
                {
                    AvoidCollisionLoading();
                }

                // If the robot can still carry more item, then it will continue to the
next shortest path location to retrieve more item
                else if (carryingCapacity - currentJob.ProdId.Weight >= 0)
                {
                    Movement(path.Item2.RetrieveCoord.Row);
                    carryingCapacity -= path.Item2.ProdId.Weight;    // update carrying
capacity

                    CarryingItem.Add(path.Item2);
                    JobList.Remove(path.Item2);    // remove item that is retrieved
                    Console.WriteLine("Product Retrieved: " +
path.Item2.ProdId.ProductName + "  " + Sector.Column + "  at  " +
path.Item2.RetrieveCoord.Row + path.Item2.RetrieveCoord.Column +
path.Item2.RetrieveCoord.Shelf);
                    Console.WriteLine("Carrying Cap: " + carryingCapacity);
                }

                // If the robot reached its max carrying capacity, then it is forced to
load items to the shipping truck
                else
                {
                    AvoidCollisionLoading();
                }

                // head straight to shipping if it is the last job and load the item on
to the truck
                if (JobList.Count == 0 && CarryingItem.Count != 0)
                {
                    AvoidCollisionLoading();
                }
            }
        }

        /// <summary>
        /// Only releases one robot into the "loading dock" row at a time to avoid
collision
        /// </summary>
        private void AvoidCollisionLoading()
        {
            Movement(Warehouse.Rows);    // travel to the last row and wait for semaphore
to allow a robot to enter one at a time

            Console.WriteLine("Waiting for gate");

            // Critical section and should be thread safe
            avoidCollision.Wait();
            LoadShipment();
            avoidCollision.Release();
```

```
            // Release of critical section

            Console.WriteLine("Release");
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="restockInfo"></param>
        /// <returns></returns>
        private async Task Restock(Jobs restockInfo)
        {
            // Move to inventory truck
            Movement(Warehouse.Rows);
            avoidCollision.Wait();
            Movement(Warehouse.LoadingDockRow);
            avoidCollision.Release();

            // Move to destination to restock
            Movement(restockInfo.RestockCoord.Row);

            Console.WriteLine("Product Restocked: " + restockInfo.ProdId.ProductName + "
" + Sector.Column + " at " + restockInfo.RestockCoord.Row +
restockInfo.RestockCoord.Column + restockInfo.RestockCoord.Shelf);
            Console.WriteLine(RobotId + " restocked complete.");


            foreach(var allProd in Warehouse.AllProducts)
            {
                if (allProd.ProductID == restockInfo.ProdId.ProductID)
                {
                    allProd.Location.Add(restockInfo.RestockCoord);

                    // Update Cloud Firestore that a new product has been restocked
                    DocumentReference restock = database.Collection("All
products").Document(restockInfo.ProdId.ProductID);
                    Dictionary<string, Object> lowStockUpdate = new Dictionary<string,
object>();

                    // increment inStock by +1 in Cloud Firestore
                    //lowStockUpdate.Add("inStock", FieldValue.Increment(1));
                    lowStockUpdate.Add("coordinate", allProd.CoordToArray());
                    await restock.UpdateAsync(lowStockUpdate);

                    break;
                }
            }


            JobList.Remove(restockInfo);
        }

        /// <summary>
        /// Load the items that the robot are currently carrying and resetting the
carrying capacity to max
        /// </summary>
        private void LoadShipment()
```

```
        {
            Movement(Warehouse.LoadingDockRow);
            carryingCapacity = Warehouse.RobotCapacity;    // resetting the carrying
capacity to max

            // Add everything to the truck
            foreach (var goingToLoad in CarryingItem)
            {
                Warehouse.AddToTruck(goingToLoad);
            }

            // Testing; used to check which products got loaded to truck
            foreach (var element in Warehouse.LoadedToTruck)
            {
                Console.WriteLine(element.ProdId.ProductName);
            }

            CarryingItem.Clear();    // empty products to the truck

            Console.WriteLine("empty: " + (CarryingItem.Count == 0));

            Movement(Warehouse.Rows);    // move out of the docking zone
        }

        /// <summary>
        /// Move robot to specified location, drain battery by 10% every 6 unit movement
== 3 seconds
        /// If robot has battery of 10%, enter power saving mode and move back to the
charging dock at charging origin (row = 0) and charge
        /// </summary>
        /// <param name="productLocation"> Where the robot will move to (rows) </param>
        private void Movement(int productLocation)
        {
            int totalUnitMovement = 0;

            while (Sector.Row != productLocation)
            {
                if (Sector.Row <= productLocation)    // Move down the row
                {
                    totalUnitMovement++;
                    Sector.Row++;
                    Console.WriteLine("Position: " + Sector.Row + " , Robot id: " +
Sector.Column);
                    UpdatePositionDB().Wait();

                    Thread.Sleep(Warehouse.TravelTime);    // Simulated travel time
                }
                else                                    // Move up the row
                {
                    totalUnitMovement++;
                    Sector.Row--;
                    Console.WriteLine("Position: " + Sector.Row + " , Robot id: " +
Sector.Column);
                    UpdatePositionDB().Wait();

                    Thread.Sleep(Warehouse.TravelTime);    // Simulated travel time
                }
```

```
                if (totalUnitMovement % 3 == 0)    // 3 movements == battery life -10%
                {
                    if (!Battery.Usage())    // check if battery is 10% then return to
charging dock
                    {
                        Movement(0);    // move back to origin with power saving mode
                        Battery.Charge();
                    }
                }
            }
        }

        /// <summary>
        /// Update robot's current position real time to the Cloud Firestore
        /// </summary>
        /// <returns> It is an asynchronous task that must be waited once called
</returns>
        private async Task UpdatePositionDB()
        {
            DocumentReference updatePos = database.Collection("All
robot").Document(RobotId);
            Dictionary<string, Object> update = new Dictionary<string, Object>();

            string currentPos = Sector.Row + " " + Sector.Column + " " + Sector.Shelf;

            update.Add("coordinate", currentPos);
            update.Add("battery", Battery.BatteryLevel.ToString());

            await updatePos.UpdateAsync(update);    // Sending coordinate updates to
Cloud Firestore
        }

        /// <summary>
        /// Find the shortest path based on the robot's current coordinates to its next
destination
        /// Always priotitize restocking jobs over retrieving jobs.
        /// </summary>
        /// <param name="allJobs"> List of products that the robot need to retrieve
</param>
        /// <returns> Returns a Tuple (int closestPath from current position to next
destination and the corresponding Job at the destination) </returns>
        public (int, Jobs) ShortestPathCalc(List<Jobs> allJobs)
        {
            int closestPath = Warehouse.LoadingDockRow - 1;
            int index = -1;
            Jobs retreval = new Jobs();

            foreach (var toRetrieve in allJobs)
            {
                // Robot prioritizes restocking over retrieving
                if (toRetrieve.Restock)
                {
                    return (-1, toRetrieve);
                }
                else
                {
```

```
                    int destination = Math.Abs(toRetrieve.RetrieveCoord.Row -
Sector.Row);

                    if (destination <= closestPath)
                    {
                        closestPath = destination;    // find the closest destination
                        retreval = toRetrieve;
                    }
                    index++;
                }
            }
            return (closestPath, retreval);
        }
    }
}
```

Trucks, ShippingTrucks, InventoryTrucks

```
using System;
using System.Collections.Concurrent;
using System.Threading;

namespace AmazoomDebug
{
    class Trucks
    {
        private double truckVol = Warehouse.TruckCapacityVol;
        private double truckWeight = Warehouse.TruckCapacityWeight;
        public static double carryVol = 0;
        public static double carryWeight = 0;
        public bool IsAvailable { get; set; } = true;
        public string TruckId { get; set; }

        public Trucks(string id)
        {
            TruckId = id;
        }

        /// <summary>
        /// Check if it is possible to load more products to the truck according to the
truck's max weight capacity and max volume capacity
        /// </summary>
        /// <param name="toLoad"> Product to be checked and loaded to the truck </param>
        /// <returns> Return true if the porduct can successfully be loaded. Otherwise
return false </returns>
        public bool LoadProduct(Products toLoad)
        {
            if (IsAvailable)
            {
                if (carryVol + toLoad.Volume <= truckVol && carryWeight + toLoad.Weight
<= truckWeight)
                {
                    carryVol += toLoad.Volume;
                    carryWeight += toLoad.Weight;
                    Console.WriteLine("{0} is loaded to {1}", toLoad.ProductName,
TruckId);
                    return true;
                }
            }
```

```
            return false;
        }

        /// <summary>
        /// Stimulates a random interval that signals the return of the truck
        /// The truck will be gone between 3 seconds to 7 seconds
        /// </summary>
        protected void NotifyArrival()
        {
            Random simulatedDeliveryTime = new Random();
            int resetTime = simulatedDeliveryTime.Next(3000, 7000);    // randomize a
time between 5 sec and 7 sec for a full cycle travel

            Thread.Sleep(resetTime);    // Simulate a random travel time for each truck
to leave and return to the warehouse
        }
    }

    /// <summary>
    /// Shipping Truck class inherited from the Truck class
    /// Ships all the products and orders that are loaded to the shipping truck
    /// </summary>
    class ShippingTruck : Trucks
    {
        public ShippingTruck(string id) : base(id) { }

        /// <summary>
        /// Load items to truck (if any) once it is allowed to dock. Leave the once the
truck is full and relases the dock for the next truck to dock.
        /// </summary>
        public void Deploy()
        {
            while (true)
            {
                Warehouse.dockLocking.Wait();
                Console.WriteLine("IsAvailable: " + IsAvailable + " TruckID: " +
TruckId);

                if (IsAvailable)
                {
                    // Testing to see if the correct truck enters the docking area
                    Console.WriteLine("{0} waiting.................................",
TruckId);

                    int loadCount = Warehouse.LoadedToTruck.Count;

                    for (int i = 0; i < loadCount; i++)
                    {
                        Warehouse.LoadedToTruck.TryDequeue(out Jobs current);
                        if (LoadProduct(current.ProdId) == false)
                        {
                            carryVol = 0;
                            carryWeight = 0;

                            Console.WriteLine("Alert!!!! SHIPPING TRUCK IS
FULL!!!!!!!!!!!!!!!!");
```

```
                        IsAvailable = false;
                        break;
                    }
                }

                // Stimulate the time it needs for the truck driver to confirm and
grant permission to commence the delivery
                if (IsAvailable == true)
                {
                    Thread.Sleep(5000);
                    IsAvailable = false;
                }

                // Release the docking area to other trucks that are waiting
                Warehouse.waitDocking.Release();
                Console.WriteLine("Truck relasing dock");

                Console.WriteLine("{0} leaving.....................................",
TruckId);

                NotifyArrival();    // Random return to the waiting area and notify
the central computer that the truck is available for use.
                IsAvailable = true;
            }
        }
    }
}

/// <summary>
/// Inventory Truck class inherited from the Truck class
/// Docks at the docking area with the products to be restocked by the robots
/// </summary>
class InventoryTruck : Trucks
{
    public ConcurrentQueue<Products> ItemInTruck { get; set; } = new
ConcurrentQueue<Products>();
    public InventoryTruck(string id) : base(id) { }
    public bool IsReady { get; set; } = false;

    /// <summary>
    /// Inventory truck starts waiting and idle
    /// Once restocking intructions are received, it proceeds to enter the docking
area and unload items in the truck
    /// It releases the docking area once it is done
    /// </summary>
    public void Deploy()
    {
        while (true)
        {
            IsReady = false;
            carryVol = 0;
            carryWeight = 0;

            if (Warehouse.RestockItem.Count != 0)
            {

                int LoadRestockToTruck = Warehouse.RestockItem.Count;
```

```
                    for (int i = 0; i < LoadRestockToTruck; i++)
                    {
                        Warehouse.RestockItem.TryDequeue(out Products currentProduct);
                        if (LoadProduct(currentProduct) == false)
                        {
                            Console.WriteLine("Alert!!!! INVENTORY TRUCK IS
FULL!!!!!!!!!!!!!!!!");
                            break;
                        }
                        else
                        {
                            ItemInTruck.Enqueue(currentProduct);
                        }
                    }

                    IsReady = true;
                    Warehouse.dockLocking.Wait();
                    Console.WriteLine("IsAvailable: " + IsAvailable + " TruckID: " +
TruckId);

                    Warehouse.createRestockingJob.Wait();

                    //TODO: Check ItemInTruck.Count loop!!!
                    while(ItemInTruck.Count != 0)
                    {
                        Thread.Sleep(2000);
                        ItemInTruck.TryDequeue(out Products discard);
                        Console.WriteLine("One Product Unloaded from inventory truck");
                    }

                    Console.WriteLine("No more item in truck");
                    Warehouse.waitDocking.Release();
                }
                else
                {
                    Thread.Sleep(5000);
                }
            }
        }
    }
}
```

Warehouse

```
using System.Collections.Concurrent;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Google.Cloud.Firestore;
using System.Threading;
using System.IO;


namespace AmazoomDebug
{
    /// <summary>
```

```
    /// contains all the global constant for the entire system
    /// Warehouse central computer system that communicates with the Cloud Firestore, all
the robots to assign jobs to, and the trucks going in and out of the warehouse
    /// </summary>
    class Warehouse
    {
        private static readonly Object addingOrderLock = new Object();
        private static readonly Object toggleWarehouseSpaceLock = new Object();
        private static readonly Object addingJobs = new Object();

        public static SemaphoreSlim dockLocking = new SemaphoreSlim(0);
        public static SemaphoreSlim waitDocking = new SemaphoreSlim(0);
        public static SemaphoreSlim invTruckLoading = new SemaphoreSlim(0);
        public static SemaphoreSlim createRestockingJob = new SemaphoreSlim(0);

        public static int LoadingDockRow { get; set; }
        public static int Rows { get; set; }
        public static int Columns { get; set; }
        public static int TravelTime { get; set; }
        public static int RobotCapacity { get; set; }
        public static double TruckCapacityVol { get; set; }
        public static double TruckCapacityWeight { get; set; }
        public static int Shelves { get; set; }
        public static ConcurrentQueue<Jobs> LoadedToTruck { get; set; } = new
ConcurrentQueue<Jobs>();
        public static ConcurrentQueue<Products> RestockItem { get; set; } = new
ConcurrentQueue<Products>();
        public static List<Products> AllProducts { get; set; } = new List<Products>();
        public static List<Jobs> AllJobs { get; set; } = new List<Jobs>();
        public static List<Orders> LocalOrders { get; set; } = new List<Orders>();

        private static List<Coordinate> isEmpty = new List<Coordinate>();
        private static List<Coordinate> isOccupied = new List<Coordinate>();
        private static List<Coordinate> accessibleLocations = new List<Coordinate>();

        private static List<Robot> operationalRobots = new List<Robot>();
        private static List<ShippingTruck> operationalShippingTrucks = new
List<ShippingTruck>();
        public static List<InventoryTruck> operationalInvTrucks = new
List<InventoryTruck>();

        private static Dictionary<string, int> partialOrders = new Dictionary<string,
int>();

        private readonly double carryWeight;
        private readonly double carryVol;

        /// <summary>
        /// Constructor
        /// Prompts for a PIN code (123123 for the purpose of this demo) and the admin
have 3 attempts
        /// Reads setup file and initializes the warehouse with all the primary global
constants when a new Warehouse Object is instantiated
        /// Ask for admin authentication and file path
        /// </summary>
        public Warehouse()
        {
```

```
            bool loggedIn = false;

            for(int trial = 3; trial > 0; trial--)
            {
                Console.WriteLine("Please enter authentication PIN:");
                int pinCode = Convert.ToInt32(Console.ReadLine());

                if (pinCode == 123123)
                {
                    Console.WriteLine("PIN Correct");
                    Console.WriteLine("Enter a file path:
(InitializationSetup/Setup_Warehouse.txt)");
                    string filePath = Console.ReadLine();

                    // read file to initialize warehouse
                    System.IO.StreamReader setup = new System.IO.StreamReader(filePath);
                    string line = setup.ReadLine();

                    // #rows, #columns, #shelves, robotCapacity, travelTime
                    string[] keys = line.Split(' ');

                    // Catch and print an error message if the file is inaccessible
                    try
                    {
                        Rows = Int32.Parse(keys[0]);                    //5
                        Columns = Int32.Parse(keys[1]);                 //8
                        Shelves = Int32.Parse(keys[2]);                 //6
                        RobotCapacity = Int32.Parse(keys[3]);           //30
                        TravelTime = Int32.Parse(keys[4]);              //1000
                        LoadingDockRow = Rows + 1;
                        TruckCapacityVol = Int32.Parse(keys[5]);        //10
                        TruckCapacityWeight = Int32.Parse(keys[6]);     //200

                        carryVol = TruckCapacityVol;
                        carryWeight = TruckCapacityWeight;

                        loggedIn = true;
                        setup.Close();

                        Console.WriteLine("Setup complete!");
                    }
                    catch
                    {
                        setup.Close();
                        Console.WriteLine("File not loaded properly and warehouse cannot
be instantiated");
                    }
                    break;
                }
                else
                {
                    Console.WriteLine("Wrong PIN. {0} attempts left", trial - 1);
                }
            }

            if (loggedIn == false)
            {
```

```
                Console.WriteLine("Set up failed. Please restart the system");
            }
        }

        /// <summary>
        /// Generate warehouse layout, robots, trucks, and products location.
        /// Product information and location are fetched from Cloud Firestore
        ///
        /// Run 2 threads to listen for updates from Cloud Firestore and are only
activated when changes have been made to the database
        /// Run 1 thread for regulating docking area so only one truck can enter at a
time
        /// Run 1 thread for sending updates of user's orders and its shipping status to
Cloud Firestore
        /// </summary>
        public void Deploy()
        {
            // Initialization of Automated Warehouse
            GenerateLayout();

            // Established a connection to Cloud Firestore with a unique application id
            string path = AppDomain.CurrentDomain.BaseDirectory + @"amazoom-c1397-
firebase-adminsdk-ho7z7-6572726fc6.json";
            Environment.SetEnvironmentVariable("GOOGLE_APPLICATION_CREDENTIALS", path);
            FirestoreDb database = FirestoreDb.Create("amazoom-c1397");

            InstantiateRobots(database).Wait();
            InstantiateTrucks(2, 1);
            FetchData(database).Wait();

            // Deploying robots
            Task[] robots = new Task[operationalRobots.Count];

            int index = 0;
            foreach (var opRobot in operationalRobots)
            {
                robots[index] = Task.Run(() => opRobot.Deploy());
                index++;
            }

            // Deploying shipping and invertory trucks
            Task[] shippingTrucks = new Task[operationalShippingTrucks.Count];
            Task[] inventoryTruck = new Task[operationalShippingTrucks.Count];

            int tIndex = 0;
            int sIndex = 0;
            foreach (var opShipTruck in operationalShippingTrucks)
            {
                shippingTrucks[sIndex] = Task.Run(() => opShipTruck.Deploy());
                sIndex++;
            }
            foreach (var opInvTruck in operationalInvTrucks)
            {
                inventoryTruck[tIndex] = Task.Run(() => opInvTruck.Deploy());
                tIndex++;
            }
```

```
            // Check for incoming order in the background and assign jobs to the robots
all in the background and adding tasks to the robot
            Task orderCheck = Task.Run(() => OrderListener(database));

            // Automatic low stock alert
            Task restockingCheck = Task.Run(() => RestockingVerification(database));

            // Check for truck loading
            Task shippingCheck = Task.Run(() => ShippingVerificationV2(database));

            Task loadToTruck = Task.Run(() => LoadingToTruck());

            // Wait all
            Task.WaitAll(robots);
            Task.WaitAll(shippingTrucks);
            Task.WaitAll(inventoryTruck);
            orderCheck.Wait();
            shippingCheck.Wait();
            loadToTruck.Wait();
            restockingCheck.Wait();
        }

        /// <summary>
        /// Generate layout based on the given information from the setup file
        /// Store all accessible coordinates into a list called accessibleLocations
        /// </summary>
        private void GenerateLayout()
        {
            // Instantiate all Coordinate Location inside the warehouse
            for(int row = 1; row <= Rows; row++)
            {
                for (int col = 1; col <= Columns; col++)
                {
                    for(int shelf = 1; shelf <= Shelves; shelf++)
                    {
                        for(int orientation = 1; orientation <= 2; orientation++)
                        {
                            Coordinate generateLayout = new Coordinate(row, col, shelf,
orientation);
                            accessibleLocations.Add(generateLayout);
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Instantiating shipping and inventory trucks
        /// </summary>
        /// <param name="shipQuantity"> number of shipping trucks to be
instantiated</param>
        /// <param name="invQuantity"> number of inventory trucks to be
instantiated</param>
        private void InstantiateTrucks(int shipQuantity, int invQuantity)
        {
            for(int i = 0; i < shipQuantity; i++)
            {
```

```
                operationalShippingTrucks.Add(new ShippingTruck("ShipTruck_" + i));
            }
            for(int i = 0; i < invQuantity; i++)
            {
                operationalInvTrucks.Add(new InventoryTruck("InvTruck_" + i));
            }
        }

        /// <summary>
        /// Instantiate equal number of robots as the number of columns in the warehouse.
Assign each robot to one column.
        /// </summary>
        /// <param name="database"> Firestore database instance </param>
        /// <returns> It is an asynchronous task that must be waited once called
</returns>
        private async Task InstantiateRobots(FirestoreDb database)
        {
            // Catch and print errors regarding database connection
            try
            {
                Console.WriteLine("Robot Instantiation");

                // Check firestore for previously initialized robots and Instantiate
robots based on previously saved locations
                Query allRobot = database.Collection("All robot");
                QuerySnapshot fetchedRobot = await allRobot.GetSnapshotAsync();

                if (fetchedRobot.Count != 0)
                {
                    foreach (DocumentSnapshot robotInfo in fetchedRobot.Documents)
                    {
                        Dictionary<string, Object> robotDetail =
robotInfo.ToDictionary();

                        string[] fetchedCoordinate =
robotDetail["coordinate"].ToString().Split(" ");

                        operationalRobots.Add(new Robot(
                            robotInfo.Id,
                            new Battery(Convert.ToInt32(robotDetail["battery"])),
                            new Coordinate(Convert.ToInt32(fetchedCoordinate[0]),
Convert.ToInt32(fetchedCoordinate[1]))
                            ));
                    }
                    Console.WriteLine("Robots info fetched sucessfully.");
                }
                else
                {
                    // Instantiating new robots
                    for (int i = 1; i <= Columns; i++)
                    {
                        operationalRobots.Add(new Robot("AMAZOOM_AW_" + i.ToString(), new
Battery(100), new Coordinate(0, i)));
                    }

                    // Updating robots information to Cloud Firestore with a unique id
for each robot at 100% battery and starting at the origin
```

```
                    int docId = 1;
                    foreach (var robots in operationalRobots)
                    {
                        DocumentReference addRobot = database.Collection("All
robot").Document("AMAZOOM_AW_" + docId.ToString());
                        Dictionary<string, string> initialRobotParams = new
Dictionary<string, string>();

                        initialRobotParams.Add("battery", "100");
                        initialRobotParams.Add("coordinate",
robots.Sector.CoordToString());

                        await addRobot.SetAsync(initialRobotParams);
                        docId++;
                    }
                    Console.WriteLine("Robot added to database successfully.");
                }
            }
            catch (Exception error)
            {
                Console.WriteLine(error);
            }
        }

        /// <summary>
        /// Initial fetching of product information from Cloud Firestore
        /// If there is no information from Cloud Firestore, then the warehouse
automatically generate 6 default products and randomly distribute them within the
warehouse
        /// The newly generated products will then be updated to Cloud Firestore
        /// </summary>
        /// <param name="database"> Firestore database instance </param>
        /// <returns> It is an asynchronous task that must be waited once called
</returns>
        private async Task FetchData(FirestoreDb database)
        {
            // Catch and print errors regarding database connection
            try
            {
                Query allProducts = database.Collection("All products");
                QuerySnapshot fetchedData = await allProducts.GetSnapshotAsync();

                // Use product information from Cloud Firestore if they exist
                if (fetchedData.Count != 0)
                {
                    foreach(DocumentSnapshot productInfo in fetchedData.Documents)
                    {
                        Dictionary<string, Object> prodDetail =
productInfo.ToDictionary();

                        List<Coordinate> assignCoord = new List<Coordinate>();
                        List<Object> fetchedCoordinates = (List<Object>)
prodDetail["coordinate"];

                        // Creating and Storing strings of coordinates from Cloud
Firestore as a Coordinate class
                        foreach(var coord in fetchedCoordinates)
```

```
                        {
                            string[] assign = coord.ToString().Split(" ");

                            // Row, Column, Shelf, RightLeft
                            Coordinate fetched = new
Coordinate(Convert.ToInt32(assign[0]), Convert.ToInt32(assign[1]),
Convert.ToInt32(assign[2]), Convert.ToInt32(assign[3]));
                            assignCoord.Add(fetched);

                            isOccupied.Add(fetched);
                        }

                        // Creating Product object for all of the documents on Cloud
Firestore
                        AllProducts.Add(new Products(
                            prodDetail["name"].ToString(),
                            productInfo.Id,
                            assignCoord,
                            Convert.ToDouble(prodDetail["weight"]),
                            Convert.ToDouble(prodDetail["volume"]),
                            Convert.ToInt32(prodDetail["inStock"]),
                            Convert.ToDouble(prodDetail["price"])));

                    }
                    Console.WriteLine("Products fetched sucessfully.");

                    // checking for empty spots in the warehouse by comparing the
isOccupied List to the default accessibleLocations list and take the differences between
the two lists
                    lock (toggleWarehouseSpaceLock)
                    {
                        foreach(var emptySpace in accessibleLocations)
                        {
                            if(isOccupied.Contains(emptySpace) == false)
                            {
                                isEmpty.Add(emptySpace);
                            }
                        }
                    }

                    foreach (var element in AllProducts)
                    {
                        Console.WriteLine(element.ProductID + " " +
element.Location[0].Row + element.Location[0].Column + element.Location[0].Shelf);
                    }
                }
                // If no product information is found on Cloud Firestore, the warehouse
automatically generate new products
                else
                {
                    InitialCoordinateRandomizer(database);    // If and only if the
warehouse is initially empty, restock with random distribution
                }
            }
            catch (Exception error)
            {
                Console.WriteLine(error);
```

```
            }
        }

        /// <summary>
        /// Randomly distribute and store products within the warehouse. Update Cloud
Firestore of each product's location
        /// </summary>
        /// <param name="database"> Firestore database instance </param>
        private void InitialCoordinateRandomizer(FirestoreDb database)
        {
            // Add new products if no data on Cloud Firestore
            List<Products> newProducts = new List<Products>()
            {
                new Products("TV", "1", 12.0, 0.373, 80, 5999.0),
                new Products("Sofa", "2", 30.0, 1.293, 80, 1250.0),
                new Products("Book", "3", 0.2, 0.005, 80, 12.0),
                new Products("Desk", "4", 22.1, 1.1, 80, 70.0),
                new Products("Phone", "5", 0.6, 0.001, 80, 1299.0),
                new Products("Bed", "6", 15, 0.73, 80, 199.0),
            };

            Random indexRandomizer = new Random();
            int totalIndex = (2 * Rows * Columns * Shelves);    // 480 for this demo
            List<int> usedIndex = new List<int>();

            lock (toggleWarehouseSpaceLock)
            {
                while(usedIndex.Count != totalIndex)
                {
                    int currentIndex = indexRandomizer.Next(totalIndex);

                    if(usedIndex.Contains(currentIndex) == false)
                    {
                        usedIndex.Add(currentIndex);
                        isEmpty.Add(accessibleLocations[currentIndex]);
                    }
                }

                // Assigning Products to a random coordinate and update to Cloud
Firestore
                foreach (var element in newProducts)
                {
                    for (int i = 1; i <= element.InStock; i++)
                    {
                        element.Location.Add(isEmpty[0]);    // Adding a new random
coordinate for the product
                        isOccupied.Add(isEmpty[0]);          // Once assigned to a
Coordinate, toggle to isOccupied
                        isEmpty.RemoveAt(0);                 // Remove the spot in
isEmpty
                    }
                }
            }
            AddProductToFirebase(database, newProducts).Wait();    // Adding product
information to Cloud Firestore
        }
```

```
        /// <summary>
        /// Asynchronously add products information to Cloud Firestore
        /// </summary>
        /// <param name="database"> Firestore database instance </param>
        /// <param name="initialProducts"> List of products to be added to Cloud
Firestore </param>
        /// <returns> It is an asynchronous task that must be waited once
called</returns>
        private async Task AddProductToFirebase(FirestoreDb database, List<Products>
initialProducts)
        {
            // Catch and print errors regarding database connection
            try
            {
                CollectionReference addingProd = database.Collection("All products");

                foreach (var prod in initialProducts)
                {
                    Dictionary<string, Object> conversion = new Dictionary<string,
object>();

                    conversion.Add("coordinate", prod.CoordToArray());    // storing all
the coordinates as a List of strings on Firestore
                    conversion.Add("inStock", prod.InStock);
                    conversion.Add("name", prod.ProductName);
                    conversion.Add("price", prod.Price);
                    conversion.Add("volume", prod.Volume);
                    conversion.Add("weight", prod.Weight);
                    conversion.Add("admin restock", 80);

                    await addingProd.AddAsync(conversion);
                }
                Console.WriteLine("Products added to database sucessfully");
            }
            catch (Exception error)
            {
                Console.WriteLine(error);
            }
        }

        /// <summary>
        /// Background listener that only gets activated once there is a change in the
"User Orders" collection on Cloud Firestore
        /// </summary>
        /// <param name="database"> Firestore database instance</param>
        private void OrderListener(FirestoreDb database)
        {
            Query incomingOrders = database.Collection("User Orders");

            FirestoreChangeListener notifier = incomingOrders.Listen(async orders =>
            {
                // TESTING: that a new order triggers this method and a new order is
stored locally: PASSED
                Console.WriteLine("New order received...");

                foreach (DocumentChange newOrders in orders.Changes)
                {
```

```
                    Dictionary<string, Object> newOrderDetail =
newOrders.Document.ToDictionary();

                    List<Object> prodInOrder = (List<Object>)newOrderDetail["Items"];
                    List<Products> tempProd = new List<Products>();

                    // Only locally store incomplete orders in the warehouse central
computer to be processed
                    if (Convert.ToBoolean(newOrderDetail["isShipped"]))
                    {
                        continue;
                    }
                    else
                    {
                        // Create Jobs and Store a copy of Orders locally
                        foreach (var prod in prodInOrder)
                        {
                            // Search id for the corresponding Product
                            foreach (var item in AllProducts)
                            {
                                if (prod.ToString() == item.ProductID)
                                {
                                    Console.WriteLine("Creating jobs...");

                                    // Creating a retrieval job
                                    Jobs newJob = new Jobs(item, newOrders.Document.Id,
false, true, item.Location[0], null);
                                    tempProd.Add(item);

                                    Console.WriteLine("item Coord: " + item.ProductName +
" " + item.Location[0].Row + item.Location[0].Column+ item.Location[0].Shelf +
item.Location[0].RightLeft);

                                    // Updating empty shelves in the warehouse
                                    lock (toggleWarehouseSpaceLock)
                                    {
                                        isEmpty.Add(item.Location[0]);
                                        isOccupied.Remove(item.Location[0]);
                                        item.Location.RemoveAt(0);
                                    }

                                    Console.WriteLine("latested Coord: " +
item.ProductName + " " + item.Location[0].Row + item.Location[0].Column +
item.Location[0].Shelf + item.Location[0].RightLeft);

                                    lock (addingJobs)
                                    {
                                        AllJobs.Add(newJob);
                                    }

                                    // TESTING: to see if new jobs is created. Product
name and retrieval coordinate should match with the one on Cloud Firestore: PASSED
                                    Console.WriteLine("New job created sucessfully... " +
newJob.ProdId.ProductName + " " + newJob.RetrieveCoord.Row + newJob.RetrieveCoord.Column
+ newJob.RetrieveCoord.Shelf + item.Location[0].RightLeft + "\nShould be assigned to
robot: " + newJob.RetrieveCoord.Column);
```

```
                                    lock (addingOrderLock)
                                    {
                                        // Instantiating incomplete jobs locally
                                        LocalOrders.Add(new Orders(
                                        newOrders.Document.Id,
                                        tempProd,
                                        newOrderDetail["user"].ToString(),
                                        Convert.ToBoolean(newOrderDetail["isShipped"])
                                        ));
                                    }

                                    break;
                                }
                            }
                        }

                        // Update Cloud Firestore of the latest coordinates for each
product. Coordinates should be removed.
                        foreach (var allProd in AllProducts)
                        {
                            DocumentReference updateStock = database.Collection("All
products").Document(allProd.ProductID);
                            Dictionary<string, Object> update = new Dictionary<string,
object>();

                            update.Add("coordinate", allProd.CoordToArray());
                            await updateStock.UpdateAsync(update);
                        }

                        // Assigning Jobs to robots by Column
                        lock (addingJobs)
                        {
                            foreach (var currentJobs in AllJobs)
                            {
                                if (currentJobs.Retrieve)
                                {
                                    // TESTING: Check if the right coordinate is assigned
to the correct robot by printing out the current coordinate. Robot from that role should
receive the job : PASSED
                                    Console.WriteLine("job location: " +
currentJobs.RetrieveCoord.Row + currentJobs.RetrieveCoord.Column +
currentJobs.RetrieveCoord.Shelf);

                                    int toAssign = (currentJobs.RetrieveCoord.Column) -
1;    // calculating product location and the corresponding robot in that columns

                                    // TESTING: Check if the right robot got assigned to
the job by printing out the assigned robot and the product name : PASSED
                                    Console.WriteLine("This job is assigned to robot: " +
toAssign + " to retrieve " + currentJobs.ProdId.ProductName);
                                    operationalRobots[toAssign].AddJob(currentJobs);
                                }
                            }
                            // Removing Jobs that are already assigned to a robot
                            AllJobs.Clear();
                        }
                    }
```

```
            }
        });

        // Thread does not end, but only get activated once there is an update in
Cloud Firestore "User Orders" collection
        notifier.ListenerTask.Wait();
    }

    /// <summary>
    /// Add item to shipping truck
    /// </summary>
    /// <param name="toTruck"> Job that contains the order id and product information
that will be loaded to the shipping truck </param>
    public static void AddToTruck(Jobs toTruck)
    {
        LoadedToTruck.Enqueue(toTruck);
    }

    /// <summary>
    /// Verify and update order status to Cloud Firestore to true if all of the
products within that order is loaded to a shipping truck
    /// </summary>
    /// <param name="database"> Firestore database instance </param>
    private async void ShippingVerificationV2(FirestoreDb database)
    {
        while (true)
        {
            // perform check against LocalOrder and update to Firebase to notify user
when every product in their order is shipped
            lock (addingOrderLock)
            {
                foreach (var loadedProduct in LoadedToTruck)
                {
                    string partialOrderId = loadedProduct.OrderId;
                    Products partialProduct = loadedProduct.ProdId;

                    foreach (var completeOrder in LocalOrders)
                    {
                        if (partialOrderId == completeOrder.OrderId)
                        {
                            if (completeOrder.Ordered.Contains(partialProduct))
                            {
                                // add new key value to dictionary, if key: orderId
and value: product count == product count in actual order, then set status to true
                                if (partialOrders.ContainsKey(completeOrder.OrderId))
                                {
                                    partialOrders[completeOrder.OrderId]--;
                                }
                                // decrement item count that is left for the
corresponding order id
                                else
                                {
                                    partialOrders.Add(completeOrder.OrderId,
completeOrder.Ordered.Count - 1);
                                }
                                completeOrder.Ordered.Remove(partialProduct);
                                break;
```

```
                            }
                        }
                    }
                }
            }

            // For each partial orders, check to see if an order is complete, if yes,
change the shipping status to true and update on to Cloud Firestore
            foreach (var pair in partialOrders)
            {
                for (int i = 0; i < LocalOrders.Count; i++)
                {
                    int emptyCount = 0;

                    // For a complete order, the dictionary containing the order id
and item count; the item count should be zero indicating that the order is complete
                    if (pair.Key == LocalOrders[i].OrderId && pair.Value ==
emptyCount)
                    {
                        DocumentReference updateOrderStatus =
database.Collection("User Orders").Document(LocalOrders[i].OrderId);
                        Dictionary<string, Object> toggleOrderStatus = new
Dictionary<string, Object>();
                        toggleOrderStatus.Add("isShipped", true);

                        // Sync to Cloud Firestore
                        await updateOrderStatus.UpdateAsync(toggleOrderStatus);

                        LocalOrders.RemoveAt(i);

                        break;
                    }
                }
            }
            // Releaseing resources for this thread. This thread does not need to
update instantly
            Thread.Sleep(2000);
        }
    }

    /// <summary>
    /// Regulates the docking area of the warehouse. Only allow one truck to enter
the docking area at a time
    /// For invertory trucks, check and create restocking orders for the
corresponding robots in the corresponding columns
    /// The restock is still randomly distributed
    /// </summary>
    private void LoadingToTruck()
    {
        while (true)
        {
            // Allowing a new truck (thread) to enter the docking area
            dockLocking.Release();
            Console.WriteLine("Warehouse releasing dock");

            // Check for restocking and inventory trucks
            foreach(var invTruck in operationalInvTrucks)
```

```
                {
                    if (invTruck.IsReady)
                    {
                        // create restocking job for the robot
                        lock (addingJobs)
                        {
                            foreach (var restockJob in invTruck.ItemInTruck)
                            {
                                lock (toggleWarehouseSpaceLock)
                                {
                                    // Updating empty shelves in the warehouse
                                    isOccupied.Add(isEmpty[0]);

                                    // Creating a new restocking job with any available
shelf unit. This will allow for a random restocking of product
                                    Jobs restock = new Jobs(restockJob, null, true,
false, null, isEmpty[0]);

                                    isEmpty.RemoveAt(0);
                                    AllJobs.Add(restock);
                                }
                            }
                            invTruck.ItemInTruck.Clear();

                            foreach(var currentJobs in AllJobs)
                            {
                                if (currentJobs.Restock)
                                {
                                    // assigning a restocking job to the corresponding
robot
                                    int toAssign = (currentJobs.RestockCoord.Column) - 1;
                                    operationalRobots[toAssign].AddJob(currentJobs);
                                }
                            }
                            AllJobs.Clear();    // clear jobs that have already been
assigned to a robot
                        }
                    }
                }
                // For inventory truck
                createRestockingJob.Release();

                // Wait for the current truck to leave the docking area first before
signalling another truck in the waiting area that it can be docked
                Console.WriteLine("Stuck waiting for release");
                waitDocking.Wait();
                Console.WriteLine("Warehouse waiting for truck to release");
            }
        }

        /// <summary>
        /// Background listener that only gets activated once there is a change in the
"All product" collection on Cloud Firestore and if the product has "inStock" of less than
60
        /// </summary>
        /// <param name="database"> Firestore database instance </param>
        private void RestockingVerification(FirestoreDb database)
```

```
        {
            // setting low stock to be 60 (for the purpose of the demo) and get an aleart
for restocking prompt
            Query checkStock = database.Collection("All
products").WhereLessThanOrEqualTo("inStock", 60);

            FirestoreChangeListener lowStockAlert = checkStock.Listen(async alert =>
            {
                foreach(var currentStock in alert.Documents)
                {
                    Dictionary<string, Object> lowAlertDict =
currentStock.ToDictionary();

                    int quantity = Convert.ToInt32(lowAlertDict["admin restock"]);
                    int inStock = Convert.ToInt32(lowAlertDict["inStock"]);

                    // Loading product to inventory truck; check weight and volume and
creating a truck
                    foreach (var allProd in AllProducts)
                    {
                        if(allProd.ProductID.Equals(currentStock.Id))
                        {
                            // loading restocking items to an available inventory truck
by the difference in the amount of stock remaining and the automatic restock setting
"admin restock"
                            for (int i = 0; i < Math.Abs(quantity-inStock); i++)
                            {
                                RestockItem.Enqueue(allProd);
                                Console.WriteLine("Restock confirmed");
                            }

                            // new dictionary for updating to Cloud Firestore
                            Dictionary<string, Object> updateStock = new
Dictionary<string, object>
                            {
                                { "inStock", quantity }
                            };

                            DocumentReference update = database.Collection("All
products").Document(currentStock.Id);

                            // Updating the stock of the product to Cloud Firestore. This
allows for user to order the same product even though the robot havent restocked the
product back to a shelf yet
                            // This is made possible because the robot prioritizes
restocking in addition to finding the shortest path
                            await update.UpdateAsync(updateStock);

                            break;
                        }
                    }
                }
            });
            // Thread does not end, but only get activated once there is an update in
Cloud Firestore "All products" collection with "inStock" of less than 60
            lowStockAlert.ListenerTask.Wait();
        }
```

```
    }
}
```