CS6310 – Software Architecture & Design
Assignment #3 [150 points]: Grocery Express Project – Individual Implementation (v4)
Fall Term 2022 – Instructor: Mark Moss

Submission Deliverables

- This assignment must be completed as an individual, not as part of a group.
- You must submit the following items in Canvas:
  (1) A ZIP file named **2022-09-A3.zip** that includes your source code. The structure of the zip file is shown in more detail below.
  (2) An updated copy of the UML Design Class Diagram from the earlier assignment reflecting any problem requirement updates as described below, named **class_diagram.pdf**.
  (3) A UML Sequence Diagram that captures the sequence of actions described below, named **sequence_diagram.pdf**.

- You must notify us via a private post on Canvas and/or Ed Discussion BEFORE the Due Date if you are encountering difficulty submitting your project. You will not be penalized for situations where Canvas is encountering significant technical problems. However, you must alert us before the due date – not well after the fact. You are responsible for submitting your answers on time in all other cases.
- Please consider that uploading files to Canvas might occasionally take a long time, even in the case of seemingly "relatively small" submissions. Plan accordingly, as submissions outside of the Canvas Availability Date will likely not be accepted. You are permitted to do unlimited submissions, so we recommend you save, upload, and submit often. You should use the same file naming standards for the (optional) "interim submissions" that you do for the final submission.

General Intent

In the first phase of the Grocery Express Project, you were required to provide design artifacts to describe your approach to structuring the classes, attributes, operations, methods and relationships needed to simulate the problem. Now, you are required to provide a lightweight implementation of the system in Java that reflects your design.

You are required to demonstrate fundamental separation of classes in your source code indicative of a reasonable design. This means that your source code should show some indication of being divided up into classes, files, etc. that match your design. You will lose points if you submit poorly structured code.

And don't panic if you realize that your original design had flaws while working on this assignment – this is common for "agile styles" of development. Make sure that your implementation works per these requirements and specifications and note your earlier design oversights with some brief comments in your code. You're also being asked to provide improved design artifacts based on your "revised and improved understanding" of the problem space, and especially in light of the problem requirement changes included in this document.

We provide some examples of the expected input and output for a few test cases. You must provide the actual Java source code. We must also be able to recompile your application from your source code as part of the evaluation process.

Problem Scenario

Your requirements for the Grocery Express Project are continued here. Any requirements from the earlier assignments are carried over here unless explicitly modified and/or otherwise cancelled. If you feel that there are any conflicts between any of the earlier requirements and the more recent requirements, then please seek clarification immediately. In general, though, the newer requirements will take precedence.

This assignment involves implementing some of the core architecture and functional capabilities for the Grocery Express Project that you designed in the earlier assignments. Your system must implement the following functionality:

(1) Create stores and allow them to offer items for purchase.
(2) Create drones that can be used by stores to deliver orders.
(3) Create pilots who can be hired by stores and assigned to fly the drones.
(4) Create customers who can start orders and request items on those orders.
(5) Allow orders to be eventually purchased or cancelled by customers.
(6) Display various reports of the system's information (e.g., stores, customers, etc.) to view the system's state and verify that the operations are working correctly.

The instructions below will provide more details about the specific functions, including input and output formats, that your system will need to implement. Your system does not have to provide a Graphical User Interface. At this point, we will evaluate your system based on the text-based output and the structure of the source code. You'll have ample opportunity to develop a more advanced interface during the group project.

Evaluation/Grading Your Submissions

- Your submission will be evaluated based on four main areas:

(1) 20 points for your system's correctness: source code provided, the capability to recompile your code, etc.
(2) 40 points for correct operation of the system on the selected scenario (test) files
(3) 50 points for reasonably structured source code
(4) 40 points for your updated Class & Sequence diagrams

Don't lose sight of the goal for the course: there are many more potential improvements that you could make to this system, but you really need to think about how you will separate and distribute the complexity of the overall problem across separate classes; and, how the objects instantiated from those classes then communicate and collaborate to solve the problem. Helping you develop a solid structure for your design is the most important aspect of working on this assignment.

(1) *20 points for your submission correctness: the capability to recompile your code, etc.*
Common issues that cause you to lose points in this category:

- The code doesn't function properly in docker during testing – for example, JNI errors, etc.
- Gross formatting errors – we've designed the testing harness to be as robust as possible when processing the submissions, but errors caused by including graphical displays of the space region, diagnostic output, and other random messages will also cause you to lose points.
- Formatting is important! Syntax is important! Use the matching characters for the output strings, and don't put extra spaces between elements of the output strings. Strings that do not match the correct output because of formatting (syntax) errors might receive significant penalties.

(2) *40 points for correct operation of the system on the selected scenario (test) files*

The test cases will be distributed across two categories: basic and advanced. The basic test cases will not cause/invoke any error conditions. The advanced test cases will likely cause one or more error conditions for each case. The score for this section will be determined by 80% of your system's success with the basic test cases + 20% of your system's success with the advanced test cases.

(3) *50 points for reasonably structured source code*
The main issue that will cause you to lose points in this category is submitting poorly structured, possibly monolithic source code that doesn't display any indication of separation of responsibilities among classes, objects, etc. There isn't a specific set of objects that you must have, but you do need to display some significant effort to apply object-oriented analysis & design principles.

(4) *40 points for your updated Class & Sequence diagrams*
The main issue that will cause you to lose points in this category is submitting Class & Sequence Diagrams that are inconsistent, and/or that don't reflect the latest changes to the problem requirements. For your Sequence Diagram, you only need to show the interactions required to successfully complete the following commands in your system:

- **`start_order`**
- **`request_item`**
- **`purchase_order`**
- **`transfer_order`**

You should create enough notional objects to demonstrate the messages and related data being shared to complete each of these operations successfully. They can be demonstrated as one connected sequence, or as individual separate sequences.

Input/Output & Command Formatting Requirements
We will test your program at this point using a relatively simple, text-based Command Line Interface (CLI) approach. The commands that we will use are listed below and your system is expected to follow the syntax of the commands as listed. Your program should accept input from a basic command (Terminal) interface – please don't use any windows, third-party text-input libraries, etc.

The basic testing is focused on the "best case/error-free" use of the system. The test cases exercise the commands to ensure that the state of the system would be updated correctly for multiple object interactions. Advanced testing includes various circumstances that should not cause updates – namely, error conditions such as undefined identifiers, duplicate identifiers, or violations of the logical (e.g., "business-level") constraints defined in the problem scenario.

For the remaining phase of the project, the system must be able to process commands that are issued with erroneous data and display an appropriate message to the given error conditions. The use case below is based on the earlier "best case/error-free" use case, while also including erroneous commands to indicate how the system must respond. The annotations below are focused on the error messages and related details.

`Welcome to the Grocery Express Delivery Service!`
The following commands demonstrate a solid use case that exercises all the required functions with some sample data. We have also provided "starter code" that highlights the required syntax for the functions to include parameters positions and data types (all Strings or Integers). The code we've provided generates the welcome header and handles the stop command to cease execution of the loop. Otherwise, you must

generate the code to handle all the remaining commands to change and display information about the state of the system.

[1] The **make_store** command must create a store with the provided name that can be used to sell items to customers, along with the store's initial revenue. The name of the store must be unique.
The "OK" message is used for commands that change the system state to acknowledge that it completed successfully.

```
> // create multiple stores
> make_store,kroger,33000
OK:change_completed
> make_store,kroger,37000
ERROR:store_identifier_already_exists
> make_store,publix,33000
OK:change_completed
```

Make stores that will sell items to customers. The store identifiers (e.g., **kroger**, **publix**) must be unique within the system.

[2] The **display_stores** command must display the information about the all the stores that have been created. Note that the "OK" message here is for commands that display information about the system state.


```
> display_stores
name:kroger,revenue:33000
name:publix,revenue:33000
OK:display_completed
```


Display the stores that have been created.

When displaying any information, **the system must display the information in ascending alphabetical order based on the identifying values used**. For example, if we created three stores with the names "whole_foods", "kroger" and "publix", then the associated records must be displayed in the order "kroger", followed by "publix", then followed by "whole_foods". This applies to all the display commands listed here and below. This is important to ensure that your program's output matches the sequence of the expected test results consistently.

There are various ways to achieve the ordering, and you are welcome to use whichever one works best for you. One suggestion: while hash maps are generally well known to most programmers and software developers, **you should also investigate Java's implementation of tree maps**.

[3] The **sell_item** command must add the item provided to the "catalog" of items available to be requested and purchased from the store. The name of the store must be valid, and the item name must be unique within that store. The weight of the item will also be provided.

```
> // create multiple items to be sold by stores
> sell_item,kroger,pot_roast,5
OK:change_completed
> sell_item,kroger,cheesecake,4
```

```
OK:change_completed
> sell_item,publix,cheesecake,8
OK:change_completed
> sell_item,whole_foods,antipasto,10
ERROR:store_identifier_does_not_exist
> sell_item,kroger,cheesecake,3
ERROR:item_identifier_already_exists
```

Make items for the store to sell. The store identifiers must be valid (i.e., defined), and the item names (e.g., **pot_roast**, **cheesecake**) must be unique within the store where they are defined.

[4] The **display_items** command must display the information about all the items that are available for request/purchase at a specific store. The information must be displayed in ascending alphabetical order of the item names.

```
> display_items,kroger
cheesecake,4
pot_roast,5
OK:display_completed
> display_items,publix
cheesecake,8
OK:display_completed
> display_items,whole_foods
ERROR:store_identifier_does_not_exist
```


Display the items for each store. The store identifiers must be valid.

[5] The **make_pilot** command must create a pilot who could fly a drone later to support grocery deliveries. The pilot must have an account (i.e., **ffig8**) that is unique among all pilots, along with a first and last name, phone number, tax ID, license ID and experience level measured in terms of the number of deliveries completed successfully. The license ID (i.e., **panam_10**) must also be unique among all pilots.


```
> // create multiple pilots to control the drones
> make_pilot,ffig8,Finneas,Fig,888-888-8888,890-12-3456,panam_10,33
OK:change_completed
> make_pilot,ggrape17,Gillian,Grape,999-999-9999,234-56-7890,twa_21,31
OK:change_completed
> make_pilot,ffig8,Frances,Faro,777-777-7777,678-90-1234,eastern_6,36
ERROR:pilot_identifier_already_exists
> make_pilot,kkiwi23,Kiara,Kiwi,555-555-5555,890-12-3456,panam_10,28
ERROR:pilot_license_already_exists
```

Make pilots to fly the drones in the system. The pilot's identifiers (e.g., **ffig8**, **ggrape17**) must be unique within the system.

[6] The **display_pilots** command must display the information about all the pilots who've been introduced in the system. The information must be displayed in ascending alphabetical order of the pilot's identifiers (e.g., ffig8).

```
> display_pilots
name:Finneas_Fig,phone:888-888-8888,taxID:890-12-3456,licenseID:panam_10,
experience:33
name:Gillian_Grape,phone:999-999-9999,taxID:234-56-7890,licenseID:twa_21,
experience:31
OK:display_completed
```

Display the status of the pilots within the system.

[7] The **make_drone** command must create a drone that can be used to deliver groceries to the appropriate customer when an order has been purchased. The drone's identifier (i.e., **1**) must be unique for the drones at that store. Drones will also have a weight capacity and a measure of the number of deliveries that the drone can make before needing refueling.

```
> // create multiple drones to deliver the orders
> make_drone,kroger,1,40,1
OK:change_completed
> make_drone,whole_foods,1,40,3
ERROR:store_identifier_does_not_exist
> make_drone,kroger,1,100,1
ERROR:drone_identifier_already_exists
> make_drone,publix,1,40,3
OK:change_completed
> make_drone,kroger,2,20,3
OK:change_completed
```


Make drones that are used to deliver the groceries. The store identifiers must be valid, and the drone identifiers (e.g., **1**, **2**) must be unique within the store where they are defined.

[8] The **display_drones** command must display the information about all the drones that can be used to deliver grocery orders for a specific store. Note that a drone will not include the name of the pilot unless one has been assigned. The information must be displayed in ascending alphabetical order of the drone's identifiers (e.g., 1). A drone's identifier is not limited to integers – it could be a string.

```
> display_drones,kroger
droneID:1,total_cap:40,num_orders:0,remaining_cap:40,trips_left:1
droneID:2,total_cap:20,num_orders:0,remaining_cap:20,trips_left:3
OK:display_completed
> display_drones,publix
droneID:1,total_cap:40,num_orders:0,remaining_cap:40,trips_left:3
OK:display_completed
> display_drones,whole_foods
ERROR:store_identifier_does_not_exist
```

Display the status of the drones for a given store. The store identifiers must be valid.

[9] The **fly_drone** command must assign the given pilot (i.e., **ffig8**) to take control of the given drone (i.e., drone **1** at the store **kroger**). A pilot can only control one drone at a time, and a drone can only be controlled by one pilot at a time.

```
> fly_drone,kroger,1,ffig8
OK:change_completed
> fly_drone,whole_foods,1,ggrape17
ERROR:store_identifier_does_not_exist
> fly_drone,publix,2,ggrape17
ERROR:drone_identifier_does_not_exist
> fly_drone,publix,1,hhoneydew20
ERROR:pilot_identifier_does_not_exist
> fly_drone,publix,1,ggrape17
OK:change_completed
```

Assign pilots to command/fly the drones. The store, drone and pilot identifiers must all be valid. Remember that a drone can only be flown/commanded by one pilot at a time, and a pilot can only fly/command one drone at a time, which has two significant implications. Suppose pilotA is being assigned to fly droneX. In this case: (1) if pilotA is currently flying a different droneY, then droneY will be left without a pilot; and, (2) if droneX is currently being flown by a different pilotB, then pilotB will be replaced by pilotA, and pilotB won't fly a drone again until they are reassigned.

[10] The **display_drones** command has already been introduced. Note that the listing now includes the name of the pilot who was recently assigned (i.e. **flown_by:Finneas_Fig**)


```
> display_drones,kroger
droneID:1,total_cap:40,num_orders:0,remaining_cap:40,trips_left:1,flown_by
:Finneas_Fig
droneID:2,total_cap:20,num_orders:0,remaining_cap:20,trips_left:3
OK:display_completed
> display_drones,publix
droneID:1,total_cap:40,num_orders:0,remaining_cap:40,trips_left:3,flown_by
:Gillian_Grape
OK:display_completed
> display_drones,whole_foods
ERROR:store_identifier_does_not_exist
```


Display the status of the drones for a given store. The store identifiers must be valid, and the drone information must include the pilot's name (e.g., **Finneas Fig**, **Gillian Grape**) if (and only if) a pilot has been assigned.

[11] The **make_customer** command must create a customer who can start orders, request items and eventually purchase (or cancel) those orders. The customer must have an account (i.e., **aapple2**) that is unique among all customers, along with a first and last name, phone number, customer rating and credits (i.e., money used to purchase items).

```
> // create multiple customers to purchase items
> make_customer,aapple2,Alana,Apple,222-222-2222,4,100
OK:change_completed
> make_customer,aapple2,Ariana,Asparagus,333-333-3333,5,150
ERROR:customer_identifier_already_exists
```

```
> make_customer,ccherry4,Carlos,Cherry,444-444-4444,5,300
OK:change_completed
```

Make customers to purchase items from stores. Customer identifiers (e.g., **aapple2**, **ccherry4**) must be unique within the system.

[12] The **display_customers** command displays all the customers who have been introduced in the system. The information must be displayed in ascending alphabetical order of the customer's identifiers (e.g., **aapple2**).

```
> display_customers
name:Alana_Apple,phone:222-222-2222,rating:4,credit:100
name:Carlos_Cherry,phone:444-444-4444,rating:5,credit:300
OK:display_completed
```

Display the status of the customers within the system.

[13] The **start_order** command must create the initial "stub" for an order at a given store for a given customer (i.e., **aapple2**). The identifier (i.e., **purchaseA**) must be unique for all orders at that store. Also, the drone and customer identifiers must be valid.

```
> // create multiple orders as requested by customers
> start_order,kroger,purchaseA,1,aapple2
OK:change_completed
> start_order,kroger,purchaseB,1,aapple2
OK:change_completed
> start_order,whole_foods,purchaseC,1,aapple2
ERROR:store_identifier_does_not_exist
> start_order,kroger,purchaseA,1,aapple2
ERROR:order_identifier_already_exists
> start_order,kroger,purchaseC,3,aapple2
ERROR:drone_identifier_does_not_exist
> start_order,kroger,purchaseC,1,bbanana3
ERROR:customer_identifier_does_not_exist
> start_order,kroger,purchaseD,2,ccherry4
OK:change_completed
> start_order,publix,purchaseA,1,ccherry4
OK:change_completed
```


Allow the customers to place orders with a store, where the order will be delivered by the designated drone. The store, drone and customer identifiers must all be valid. And the order identifier must be unique within the store where the order has been placed.

[14] The **display_orders** command must display information about all the orders at a given store. The orders should also display all the items that have been requested for each order. The information must be displayed in ascending alphabetical order of the store order's identifiers (e.g., **purchaseA**).

```
> display_orders,kroger
orderID:purchaseA
orderID:purchaseB
```

```
orderID:purchaseD
OK:display_completed
> display_orders,publix
orderID:purchaseA
OK:display_completed
> display_orders,whole_foods
ERROR:store_identifier_does_not_exist
```

Display the orders that have been placed with a given store. The store identifiers must be valid. Also, the contents of the orders must be displayed as well. Though all of the orders are blank here – because the customers haven't requested any items yet – later executions of this command will demonstrate what the orders look like when items have been added.

[15] The **request_item** command must add an item to the designated order. The quantity and unit price (i.e., **3** and **9**, respectively) will be decided at the time the command is entered. An item can only be added to an order once. More importantly, the item must be added to the order if – and only if – the following conditions are true: (1) the customer has enough remaining credits to afford the new item; and, (2) the drone has enough remaining capacity to carry the new item as part of its payload.

```
> // add multiple items to the orders
> request_item,kroger,purchaseA,pot_roast,3,10
OK:change_completed
> request_item,kroger,purchaseB,pot_roast,4,5
OK:change_completed
> request_item,whole_foods,purchaseA,cheesecake,1,10
ERROR:store_identifier_does_not_exist
> request_item,kroger,purchaseE,cheesecake,1,10
ERROR:order_identifier_does_not_exist
> request_item,kroger,purchaseA,truffle_risotto,1,10
ERROR:item_identifier_does_not_exist
> request_item,kroger,purchaseA,pot_roast,1,10
ERROR:item_already_ordered
> request_item,kroger,purchaseA,cheesecake,1,90
ERROR:customer_cant_afford_new_item
> request_item,kroger,purchaseA,cheesecake,10,5
ERROR:drone_cant_carry_new_item
> request_item,publix,purchaseA,cheesecake,3,10
OK:change_completed
> request_item,kroger,purchaseD,cheesecake,1,10
OK:change_completed
```

Customers request that items sold in the given stores be added to their orders. Though the weight of an item is designated as part of the catalog of items offered by the store, the quantity and price are determined when this command is executed. The store, order and item identifiers must be valid. And an item can be added to an order only once. Finally, per the problem description, the drone must be able to carry the weight of all of the items in the orders being carried, and the customer must have enough credits to afford all of the items that they have requested so far.

[16] The **display_orders** command has already been introduced. Note that the display now includes information about all the items listed under each order.

```
> // display the state of the simulation
> display_orders,kroger
orderID:purchaseA
item_name:pot_roast,total_quantity:3,total_cost:30,total_weight:15
orderID:purchaseB
item_name:pot_roast,total_quantity:4,total_cost:20,total_weight:20
orderID:purchaseD
item_name:cheesecake,total_quantity:1,total_cost:10,total_weight:4
OK:display_completed
> display_orders,publix
orderID:purchaseA
item_name:cheesecake,total_quantity:3,total_cost:30,total_weight:24
OK:display_completed
```

[17] All of the following commands have already been discussed. They are repeated here to help illustrate the changes in the system as orders are either purchased or cancelled by the customers.

```
> display_customers
name:Alana_Apple,phone:222-222-2222,rating:4,credit:100
name:Carlos_Cherry,phone:444-444-4444,rating:5,credit:300
OK:display_completed
> display_stores
name:kroger,revenue:33000
name:publix,revenue:33000
OK:display_completed
> display_drones,kroger
droneID:1,total_cap:40,num_orders:2,remaining_cap:5,trips_left:1,flown_by:
Finneas_Fig
droneID:2,total_cap:20,num_orders:1,remaining_cap:16,trips_left:3
OK:display_completed
> display_drones,publix
droneID:1,total_cap:40,num_orders:1,remaining_cap:16,trips_left:3,flown_by
:Gillian_Grape
OK:display_completed
> display_pilots
name:Finneas_Fig,phone:888-888-8888,taxID:890-12-3456,licenseID:panam_10,
experience:33
name:Gillian_Grape,phone:999-999-9999,taxID:234-56-7890,licenseID:twa_21,
experience:31
OK:display_completed
```


[18] The **purchase_order** command must complete the purchase of the order and the delivery of the groceries to the appropriate customer. The purchase must be completed if – and only if – the drone delivering the order has a pilot assigned and enough fuel to complete the delivery. There are five significant changes to the system's state that must be enacted to successfully complete the purchase: (1) the cost of the order must be deducted from the customer's credits; (2) the cost of the order must be added to the store's revenue; (3)

the number of remaining deliveries (i.e., fuel) for the drone must be reduced by one; (4) the pilot's experience (i.e., number of successful deliveries) must be increased by one; and, (5) the order must otherwise be removed from the system.

```
> // deliver an order and display the updated state
> purchase_order,kroger,purchaseA
OK:change_completed
```

Customers may close an order and request that it be delivered as quickly as possible. When the delivery occurs: (a) the customer pays the store for the (final) order; (b) the drone uses fuel to deliver the order; and, (c) the pilot flying the drone is credited with a successful delivery.

[19] The following commands have been displayed to demonstrate the changes to the system's state after the delivery of the **purchaseA** order. The **$30** of credits for the **purchaseA** order have been transferred from **aapple2** to **kroger**, and the drone fuel and pilot experience have been updated.

```
> display_orders,kroger
orderID:purchaseB
item_name:pot_roast,total_quantity:4,total_cost:20,total_weight:20
orderID:purchaseD
item_name:cheesecake,total_quantity:1,total_cost:10,total_weight:4
OK:display_completed
> display_orders,publix
orderID:purchaseA
item_name:cheesecake,total_quantity:3,total_cost:30,total_weight:24
OK:display_completed
> display_customers
name:Alana_Apple,phone:222-222-2222,rating:4,credit:70
name:Carlos_Cherry,phone:444-444-4444,rating:5,credit:300
OK:display_completed
> display_stores
name:kroger,revenue:33030
name:publix,revenue:33000
OK:display_completed
> display_drones,kroger
droneID:1,total_cap:40,num_orders:1,remaining_cap:20,trips_left:0,flown_by
:Finneas_Fig
droneID:2,total_cap:20,num_orders:1,remaining_cap:16,trips_left:3
OK:display_completed
> display_drones,publix
droneID:1,total_cap:40,num_orders:1,remaining_cap:16,trips_left:3,flown_by
:Gillian_Grape
OK:display_completed
> display_pilots
name:Finneas_Fig,phone:888-888-8888,taxID:890-12-3456,licenseID:panam_10,
experience:34
name:Gillian_Grape,phone:999-999-9999,taxID:234-56-7890,licenseID:twa_21,
experience:31
OK:display_completed
```

After the previous delivery, note that **$30** (the cost of the **purchaseA** order from **kroger**) has been transferred from the customer to the store, and the pilot and drone stats have also been updated.

[20] For the purchase & delivery of an order to be successful, the store and order identifiers must be valid, and the drone must have an assigned pilot and enough fuel for one more (remaining) trip.

```
> // deliver orders from various stores
> purchase_order,whole_foods,purchaseA
ERROR:store_identifier_does_not_exist
> purchase_order,kroger,purchaseF
ERROR:order_identifier_does_not_exist
> purchase_order,kroger,purchaseB
ERROR:drone_needs_fuel
> purchase_order,kroger,purchaseD
ERROR:drone_needs_pilot
```

[21] The following commands provide more examples of the effects of orders being purchased.

```
> fly_drone,kroger,2,ffig8
OK:change_completed
> purchase_order,kroger,purchaseE
OK:change_completed
> display_orders,kroger
orderID:purchaseB
item_name:pot_roast,total_quantity:4,total_cost:20,total_weight:20
OK:display_completed
> display_orders,publix
orderID:purchaseA
item_name:cheesecake,total_quantity:3,total_cost:30,total_weight:24
OK:display_completed
> display_customers
name:Alana_Apple,phone:222-222-2222,rating:4,credit:70
name:Carlos_Cherry,phone:444-444-4444,rating:5,credit:290
OK:display_completed
> display_stores
name:kroger,revenue:33040
name:publix,revenue:33000
OK:display_completed
> display_drones,kroger
droneID:1,total_cap:40,num_orders:1,remaining_cap:20,trips_left:0
droneID:2,total_cap:20,num_orders:0,remaining_cap:20,trips_left:2,flown_by
:Finneas_Fig
OK:display_completed
> display_drones,publix
droneID:1,total_cap:40,num_orders:1,remaining_cap:16,trips_left:3,flown_by
:Gillian_Grape
OK:display_completed
> display_pilots
```

```
name:Finneas_Fig,phone:888-888-8888,taxID:890-12-
3456,licenseID:panam_10,experience:35
name:Gillian_Grape,phone:999-999-9999,taxID:234-56-
7890,licenseID:twa_21,experience:31
OK:display_completed
> purchase_order,publix,purchaseA
OK:change_completed
> display_orders,kroger
orderID:purchaseB
item_name:pot_roast,total_quantity:4,total_cost:20,total_weight:20
OK:display_completed
> display_orders,publix
OK:display_completed
> display_customers
name:Alana_Apple,phone:222-222-2222,rating:4,credit:70
name:Carlos_Cherry,phone:444-444-4444,rating:5,credit:260
OK:display_completed
> display_stores
name:kroger,revenue:33040
name:publix,revenue:33030
OK:display_completed
> display_drones,kroger
droneID:1,total_cap:40,num_orders:1,remaining_cap:20,trips_left:0
droneID:2,total_cap:20,num_orders:0,remaining_cap:20,trips_left:2,flown_by
:Finneas_Fig
OK:display_completed
> display_drones,publix
droneID:1,total_cap:40,num_orders:0,remaining_cap:40,trips_left:2,flown_by
:Gillian_Grape
OK:display_completed
> display_pilots
name:Finneas_Fig,phone:888-888-8888,taxID:890-12-
3456,licenseID:panam_10,experience:35
name:Gillian_Grape,phone:999-999-9999,taxID:234-56-
7890,licenseID:twa_21,experience:32
OK:display_completed
```


[22] The **cancel_order** command must remove the order from the system without otherwise changing the system' state.

```
> // cancel orders per customer requests
> cancel_order,whole_foods,purchaseB
ERROR:store_identifier_does_not_exist
> cancel_order,kroger,purchaseG
ERROR:order_identifier_does_not_exist
> cancel_order,kroger,purchaseB
OK:change_completed
```

This is an example of cancelling an order, which simply removes the order from the system. The store and order identifiers must still be valid; however, a pilot and sufficient fuel for the assigned drone are not required for successful cancellation.

[23] The following commands display the effects of orders being cancelled. These commands have already been introduced and are displayed here just to confirm that key elements of the simulation state have not been modified after the **purchaseB** order from **kroger** was cancelled.

```
> display_orders,kroger
OK:display_completed
> display_orders,publix
OK:display_completed
> display_customers
name:Alana_Apple,phone:222-222-2222,rating:4,credit:70
name:Carlos_Cherry,phone:444-444-4444,rating:5,credit:260
OK:display_completed
> display_stores
name:kroger,revenue:33040
name:publix,revenue:33030
OK:display_completed
> display_drones,kroger
droneID:1,total_cap:40,num_orders:0,remaining_cap:40,trips_left:0
droneID:2,total_cap:20,num_orders:0,remaining_cap:20,trips_left:2,flown_by
:Finneas_Fig
OK:display_completed
> display_drones,publix
droneID:1,total_cap:40,num_orders:0,remaining_cap:40,trips_left:2,flown_by
:Gillian_Grape
OK:display_completed
> display_pilots
name:Finneas_Fig,phone:888-888-8888,taxID:890-12-
3456,licenseID:panam_10,experience:35
name:Gillian_Grape,phone:999-999-9999,taxID:234-56-
7890,licenseID:twa_21,experience:32
OK:display_completed
```


None of the store, customer, drone, or pilot stats were changed because of the cancellation.

[24] The **transfer_order** command must cause the order being referenced to be moved to a new drone controlled by the same store. The new drone must have the capacity to carry the new order. The following sequence of commands are demonstrated below. At the beginning of this portion of the simulation, drone #1 holds orders **ordA** and **ordB**; drone #2 holds **ordC**; and drone #3 holds **ordD**:

```
> display_orders,fresh_market
orderID:ordA
item_name:bacon,total_quantity:1,total_cost:5,total_weight:3
orderID:ordB
item_name:bacon,total_quantity:1,total_cost:5,total_weight:3
orderID:ordC
orderID:ordD
```

```
OK:display_completed
> display_drones,fresh_market
droneID:1,total_cap:10,num_orders:2,remaining_cap:4,trips_left:3,flown_by:
Alan_Alexander
droneID:2,total_cap:5,num_orders:1,remaining_cap:5,trips_left:0,flown_by:G
ina_Garcia
droneID:3,total_cap:6,num_orders:1,remaining_cap:6,trips_left:2,flown_by:P
eter_Parsons
OK:display_completed
```

In the following sequence, we add an item to order **ordC**, and then fail to transfer order **ordB** from drone #1 to drone #2:

```
> request_item,fresh_market,ordC,bacon,1,5
OK:change_completed
> transfer_order,fresh_market,ordB,2
ERROR:new_drone_does_not_have_enough_capacity
> display_orders,fresh_market
orderID:ordA
item_name:bacon,total_quantity:1,total_cost:5,total_weight:3
orderID:ordB
item_name:bacon,total_quantity:1,total_cost:5,total_weight:3
orderID:ordC
item_name:bacon,total_quantity:1,total_cost:5,total_weight:3
orderID:ordD
OK:display_completed
> display_drones,fresh_market
droneID:1,total_cap:10,num_orders:2,remaining_cap:4,trips_left:3,flown_by:
Alan_Alexander
droneID:2,total_cap:5,num_orders:1,remaining_cap:2,trips_left:0,flown_by:G
ina_Garcia
droneID:3,total_cap:6,num_orders:1,remaining_cap:6,trips_left:2,flown_by:P
eter_Parsons
OK:display_completed
```


In the following sequence, we add an item to order **ordD**, and then successfully transfer order **ordC** from drone #2 to drone #3:

```
> request_item,fresh_market,ordD,bacon,1,5
OK:change_completed
> transfer_order,fresh_market,ordC,3
OK:change_completed
> display_orders,fresh_market
orderID:ordA
item_name:bacon,total_quantity:1,total_cost:5,total_weight:3
orderID:ordB
item_name:bacon,total_quantity:1,total_cost:5,total_weight:3
orderID:ordC
item_name:bacon,total_quantity:1,total_cost:5,total_weight:3
orderID:ordD
item_name:bacon,total_quantity:1,total_cost:5,total_weight:3
OK:display_completed
```

```
> display_drones,fresh_market
droneID:1,total_cap:10,num_orders:2,remaining_cap:4,trips_left:3,flown_by:
Alan_Alexander
droneID:2,total_cap:5,num_orders:0,remaining_cap:5,trips_left:0,flown_by:G
ina_Garcia
droneID:3,total_cap:6,num_orders:2,remaining_cap:0,trips_left:2,flown_by:P
eter_Parsons
OK:display_completed
```

In the following sequence, we initiate three different transfers, though the second transfer – of order **ordC** – isn't really implemented because **ordC** is already being held by drone #3:

```
> transfer_order,fresh_market,ordD,1
OK:change_completed
> transfer_order,fresh_market,ordC,3
OK:new_drone_is_current_drone_no_change
> transfer_order,fresh_market,ordC,2
OK:change_completed
```

[25] The **display_effiency** command displays information about three metrics for each store: [1] the number of purchased orders that have been made from that store; [2] the number of "overloads", or extra packages that have been carried around by a drone without being delivered; and [3] the number of transfers that have been successfully executed. Overloads are calculated as the extra (i.e., more-purchased at that time) packages held by a drone when it makes a delivery. Note that we have executed three successful transfers at the beginning of this portion of the sequence:


```
> display_drones,fresh_market
droneID:1,total_cap:10,num_orders:3,remaining_cap:1,trips_left:3,flown_by:
Alan_Alexander
droneID:2,total_cap:5,num_orders:1,remaining_cap:2,trips_left:0,flown_by:G
ina_Garcia
droneID:3,total_cap:6,num_orders:0,remaining_cap:6,trips_left:2,flown_by:P
eter_Parsons
OK:display_completed
> // deliver an order from Fresh Market and display the updated state
> display_efficiency
name:fresh_market,purchases:0,overloads:0,transfers:3
OK:display_completed
```

When drone #1 makes the first delivery of order **ordA**, the drone is also holding orders **ordB** and **ordD** which count as overloads:

```
> purchase_order,fresh_market,ordA
OK:change_completed
> display_drones,fresh_market
droneID:1,total_cap:10,num_orders:2,remaining_cap:4,trips_left:2,flown_by:
Alan_Alexander
droneID:2,total_cap:5,num_orders:1,remaining_cap:2,trips_left:0,flown_by:G
ina_Garcia
droneID:3,total_cap:6,num_orders:0,remaining_cap:6,trips_left:2,flown_by:P
eter_Parsons
OK:display_completed
```

```
> display_efficiency
name:fresh_market,purchases:1,overloads:2,transfers:3
OK:display_completed
```

When drone #1 makes the next delivery of order **ordB**, the drone is still holding order **ordD** which counts as an overload:

```
> purchase_order,fresh_market,ordB
OK:change_completed
> display_drones,fresh_market
droneID:1,total_cap:10,num_orders:1,remaining_cap:7,trips_left:1,flown_by:
Alan_Alexander
droneID:2,total_cap:5,num_orders:1,remaining_cap:2,trips_left:0,flown_by:G
ina_Garcia
droneID:3,total_cap:6,num_orders:0,remaining_cap:6,trips_left:2,flown_by:P
eter_Parsons
OK:display_completed
> display_efficiency
name:fresh_market,purchases:2,overloads:3,transfers:3
OK:display_completed
```

When drone #1 makes the final delivery of order **ordD**, the drone isn't holding any other orders, so there are no additional overloads for this delivery:


```
> purchase_order,fresh_market,ordD
OK:change_completed
> display_drones,fresh_market
droneID:1,total_cap:10,num_orders:0,remaining_cap:10,trips_left:0,flown_by
:Alan_Alexander
droneID:2,total_cap:5,num_orders:1,remaining_cap:2,trips_left:0,flown_by:G
ina_Garcia
droneID:3,total_cap:6,num_orders:0,remaining_cap:6,trips_left:2,flown_by:P
eter_Parsons
OK:display_completed
> display_efficiency
name:fresh_market,purchases:3,overloads:3,transfers:3
OK:display_completed
```

[26] The **stop** command must cause the (otherwise infinite) interactive loop to halt.

```
> stop
stop acknowledged
simulation terminated
```

[27] Comments: If the command is interpreted as a comment – more specifically, if the line begins with the comment indicator "//" – then your program must print out the line contents & continue to the next input line/command.

Error Conditions and Display Priority

When executing a sequence of commands during a simulation run/test case, your system might encounter a situation where two or more different error conditions/messages could be displayed. To ensure that our

system operates consistently (i.e., deterministically for testing purposes), we've presented of the error messages for each of the operations. The error messages are listed in order of priority – if two or more messages are valid for a given operation, then list the first (i.e. highest priority) message only.

`make_store()` command
ERROR: "store_identifier_already_exists"
`sell_item()` command
ERROR: "store_identifier_does_not_exist"
ERROR: "item_identifier_already_exists"
`display_items()` command
ERROR: "store_identifier_does_not_exist"
`make_pilot()` command
ERROR: "pilot_identifier_already_exists"
ERROR: "pilot_license_already_exists"
`make_drone()` command
ERROR: "store_identifier_does_not_exist"
ERROR: "drone_identifier_already_exists"
display_drones() command
ERROR: "store_identifier_does_not_exist"
`fly_drone()` command
ERROR: "store_identifier_does_not_exist"
ERROR: "drone_identifier_does_not_exist"
ERROR: "pilot_identifier_does_not_exist"
`make_customer()` command
ERROR: "customer_identifier_already_exists"
`start_order()` command
ERROR: "store_identifier_does_not_exist"
ERROR: "order_identifier_already_exists"
ERROR: "drone_identifier_does_not_exist"
ERROR: "customer_identifier_does_not_exist"
`display_orders()` command
ERROR: "store_identifier_does_not_exist"
`request_item()` command
ERROR: "store_identifier_does_not_exist"
ERROR: "order_identifier_does_not_exist"
ERROR: "item_identifier_does_not_exist"
ERROR: "item_already_ordered"
ERROR: "customer_cant_afford_new_item"
ERROR: "drone_cant_carry_new_item"
`purchase_order()` command
ERROR: "store_identifier_does_not_exist"
ERROR: "order_identifier_does_not_exist"
ERROR: "drone_needs_pilot"
ERROR: "drone_needs_fuel"
`cancel_order()` command
ERROR: "store_identifier_does_not_exist"
ERROR: "order_identifier_does_not_exist"
`transfer_order()` command
ERROR: "store_identifier_does_not_exist"
ERROR: "order_identifier_does_not_exist "
ERROR: "drone_identifier_does_not_exist"
ERROR:
"new_drone_does_not_have_enough_capacity"
OK: "new_drone_is_current_drone_no_change"

Sample Code

We've included some sample Java source "starter code" that provides the command loop for terminal-based data input. You are also welcomed to use portions of the code in your own solution and use of the provided starter code is 100% optional: you are NOT required/obligated to use this code. We've also provided some test cases, though you are also welcome (and encouraged) to develop your own tests as well. You are permitted to share your test cases with fellow students, but you are not permitted to share code and/or specific directions on the code or designs need to pass the cases.

If you would like to take a closer look at the starter code, then the best way might be to create a new Java project with blank Main and DeliveryService files, and then import the provided content. Similarly, you can compile and experiment with the code and create a new JAR file as well. We've also provided an initial copy of the JAR file that can be executed in two ways:

[1] Interactively (Enter the commands manually as highlighted):

```
$ java -jar drone_delivery.jar
Welcome to the Grocery Express Delivery Service!
make_store,kroger,33000
```

```
> make_store,kroger,33000
OK:change_completed
display_stores
> display_stores
name:kroger,revenue:33000
OK:display_completed
...
stop
> stop
stop acknowledged
simulation terminated
$
```

[2] Scripted (Recommended: Enter the commands via a text file as highlighted):

```
$ java -jar drone_delivery.jar < test_case_0.txt
Welcome to the Grocery Express Delivery Service!
> make_store,kroger,33000
OK:change_completed
> display_stores
name:kroger,revenue:33000
OK:display_completed
...
> stop
stop acknowledged
simulation terminated
$
```


Also, the test cases that we will use for this phase of the project will be fundamentally correct and consistent and will not include any errors like duplicate references (e.g., attempting to make two stores with the same name), invalid references (e.g., referring to non-existent stores or customers), or other (logical) constraints with respect to the problem/scenario description.

Submission Details (Zip File)

- You must submit a single ZIP file with the following structure:
  - 2022-09-A3 (folder)
    - src (folder)
      - bunch of *java files (files)
        - Note: They could be at different levels / folders under src based on your setup.
- No changes to the docker file or other files will be allowed in your submission.
- The structure above is meant to replicate your local structure. Please do not exclude any files under your source directory & make sure you include all src/* files. Please also retain the proper file / folder structure under src. Example: if you have deliveryservice.java under src/edu/gatech/cs6310/deliveryservice.java, do not exclude the file from your zip & do not consolidate the file to src/deliveryservice.java if that is not where it is properly located.
- We have given a zip.sh and zip.ps1 scripts. You can read how to use them in the github readme. Please verify they work correctly and the zip you submit has everything requested.

- Your system must complete each test case in less than three (3) seconds (real time) – otherwise, your result will be considered to be a failure for that scenario. The scale of these scenarios shouldn't require more computing resources than this – in fact, many systems from the previous term completed all of the test cases (e.g., 20+ cases) in less than three (3) seconds total. Let us know if you feel that you are unable to develop your system to meet this requirement.

- Your program should display the output directly to "standard output" on the command line or terminal – not to a file, special console, etc. And you are welcome to use diagnostic output when testing and troubleshooting your program; however, you must disable or otherwise remove any "excess output" before you submit your program. Any excess (i.e. non-required) output will disrupt the automated evaluation of your system, and requests to re-evaluate your system on this basis will be penalized heavily.

- On a related note, it's highly suggested that you test your finished system on a separate machine if possible, away from the original development environment. Unfortunately, we do receive otherwise correct solutions that fail during our tests because of certain common errors:
  - forgetting to include one or more external libraries, etc.
  - having your program read files from a specific, hardcoded (and non-existent) folder
  - having your program read test files embedded files in its own JAR package

- Many modern Integrated Development Environments (IDEs) such as Eclipse, IntelliJ, Android Studio, etc. offer very straightforward features that will allow you to create a runnable JAR file fairly easily. Also, please be aware that we will recompile your source code to verify the functionality & evaluation of your solution compared to your submitted source code.

Closing Comments & Suggestions

This is the information that has been provided by the customer so far. We (the OMSCS 6310 Team) will likely conduct an Office Hours where you will be permitted to ask us questions in order to clarify the client's intent, etc. We will answer some of the questions, but we will not necessarily answer all of them. Also, though this current version of the system if "the core" of the system going forward, our clients will very likely add, update, and possibly remove some of the requirements over the span of the course. One of your main tasks will be to ensure that your architectural documents and related artifacts remain consistent with the problem requirements – and with your system implementations – over time.

Quick Reminder on Collaborating with Others

Please use Ed Discussion for your questions and/or comments, and post publicly whenever it is appropriate. If your questions or comments contain information that specifically provides an answer for some part of the assignment, then please make your post private first, and we (the OMSCS 6310 Team) will review it and decide if it is suitable to be shared with the larger class. Best of luck on to you this assignment, and please contact us if you have questions or concerns.