Ordering System

Project Report

Group Members: Jenny Goldsher, Deandra Martin, Hiroki Sato, Miriam Scheinblum

Course: Object-Oriented Design and Development

Professor: Dr. R. Elva

Due Date: April 23, 2020

Table of Contents

Introduction 1

Narrative 2

Object-Oriented Design 4

- Use Case Diagram 4
- Activity Diagrams 5
- UML Class Diagram 8

Implementation 10

- How did we code? 10
- Who did what? 11
- Test Cases 11

Reflection 12

Code 14

- Classes 14
- Test Suits 34
- Driver 52
- Input files 73

Introduction

As aspiring Software Engineers, it is our duty to understand the importance of the Software Engineering process. Designing an elegant system, such as our own, took two key things: knowledge and team effort. Our foundation of Object-Oriented Programming concepts was the fuel that kept the process smooth and helped us to personalize our strategy and understanding. In doing this, we utilized communication and exploring other's ideas to come up with the best solution to our problem.

Narrative

An online company must manage orders placed by its customers. Every order has the name of the customer, the total cost of the order (cost of each item multiplied by its quantity), an item(s), the date on which the order was placed, a status and a tracking number. An order can be either in-store pickup or shipped.

Shipped orders have a shipping cost, shipping company, and a shipping cost (10% of the total cost of item(s) ordered). If an item ordered is not currently in stock, then the order is recorded as a waitlist order and request for the item will be sent to the supplier - such orders have an expected ship date. Customers should also have the option of making an express order which incurs an express fee of $4.00 on the total of the order; it is separate from their shipping cost. In-store pickups have a location for pick up and a status, indicating if it is ready for pickup. Once an order has been delivered or picked up, the system will notify the company and then deleted it.

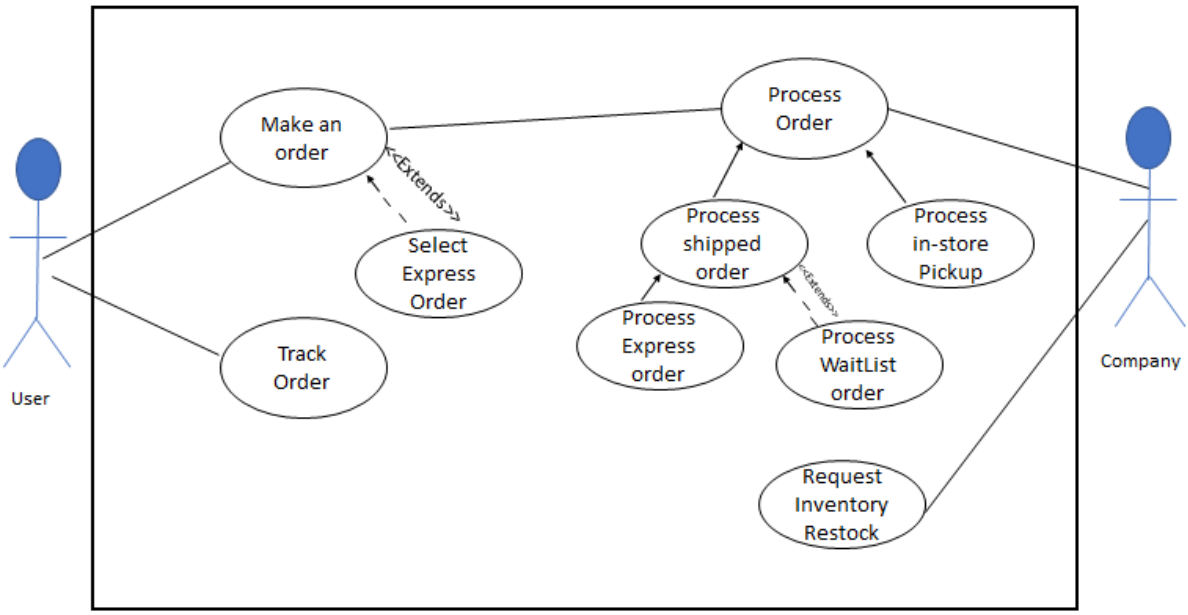
It should be possible to automatically cancel a waitlist order once it does not come back in stock within 14 days. When a user orders multiple items and one of the items is not in stock, the user will be asked to hold the order until the item is back in stock or to separate the order in a shipped and a waitlist order for the item not currently in stock.

In addition, an inventory of items shows what is available to fulfill the orders. The inventory class keeps track of the items in stock, removes items once they have been assigned to an order and adds items once items are provided by the supplier. Each item has a name, stock no., cost, and a supplier. When an order is created, inventory undergoes a check if the item is in

stock. The inventory checks automatically every 4 days for items in low numbers and automatically restocks them.

Object-Oriented Design

Use Case Diagram



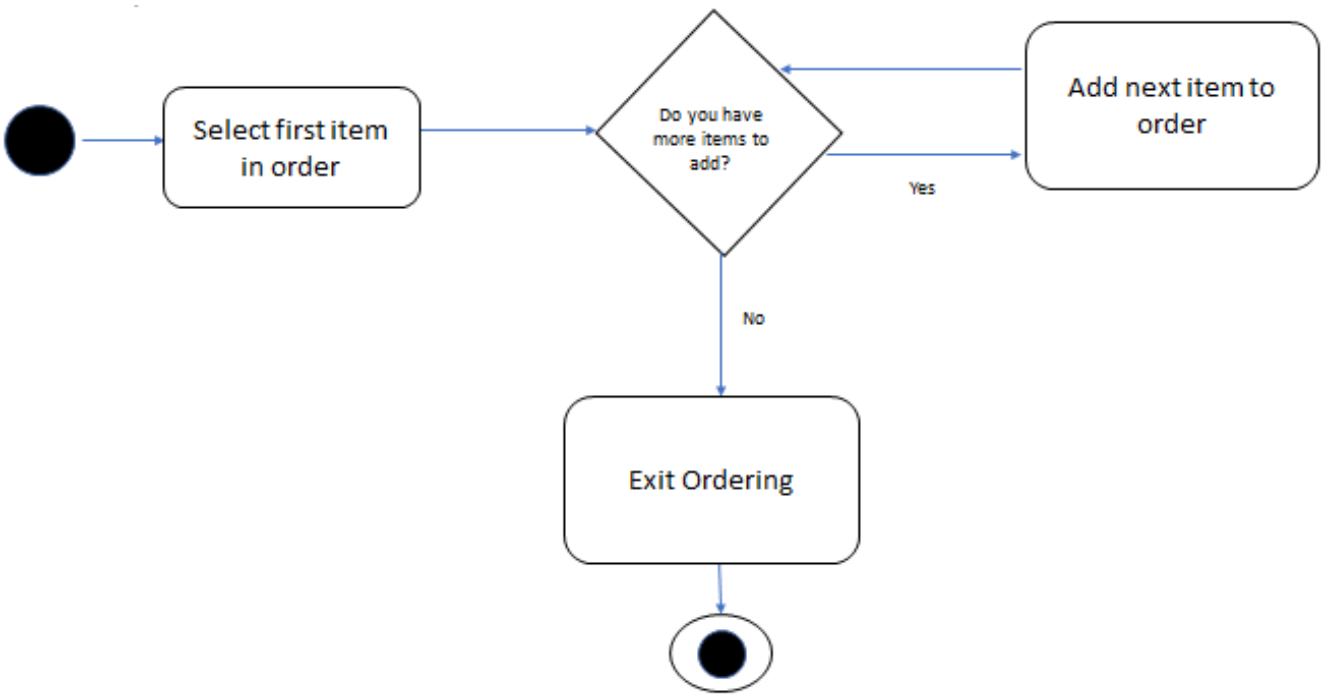
The Use Case Diagram shows how both the user and the company interact with the Ordering System. The user can make an order which includes the possibility of them designating their shipped order as expressed which has an additional fee. Users can also track their order with their tracking number. The company can use the system to process orders that have been input by users. Once the order is made by the user, orders are processed by the company through various specializations: In-Store pickup, Shipped Order, Express Order, and Waitlist order. Based on the status of the inventory, the company can request a restock from the supplier.

Activity Diagrams

The Activity diagrams represent the individual use cases of the system. They show in-depth what happens during each process from either the User or the System depending on what the activity pertains to. These models provide a good outline of how programs should be structured by giving a readable sequence of events and conditional statements.

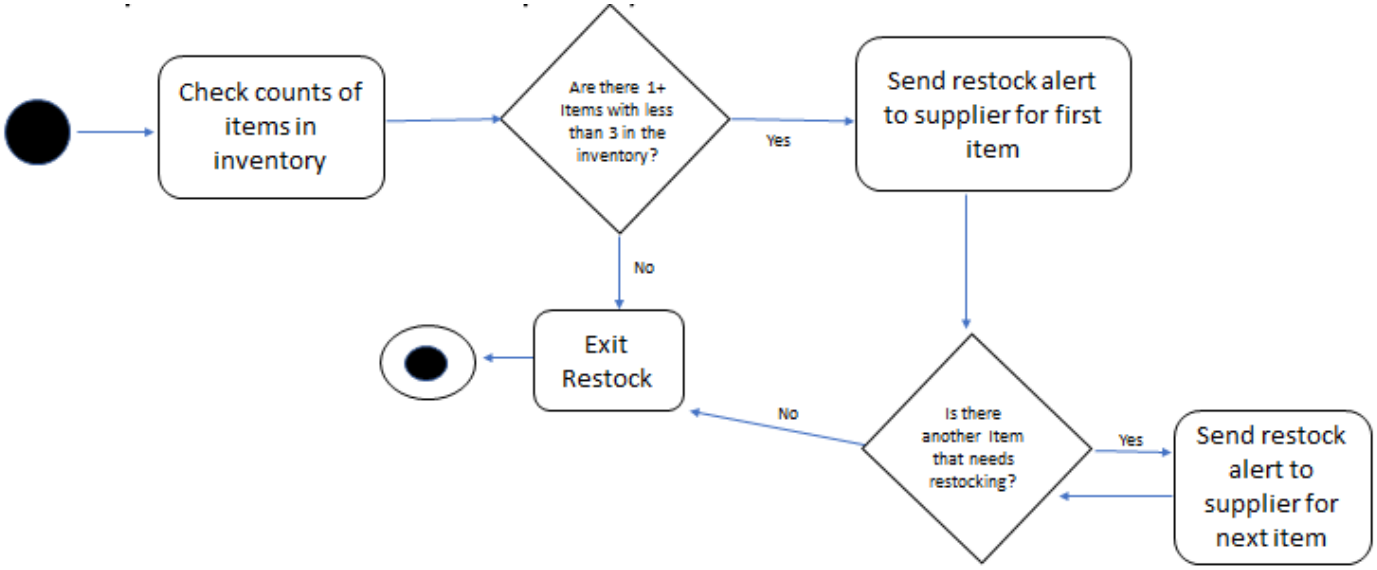
Activity Diagram: Make an Order

Perspective: User



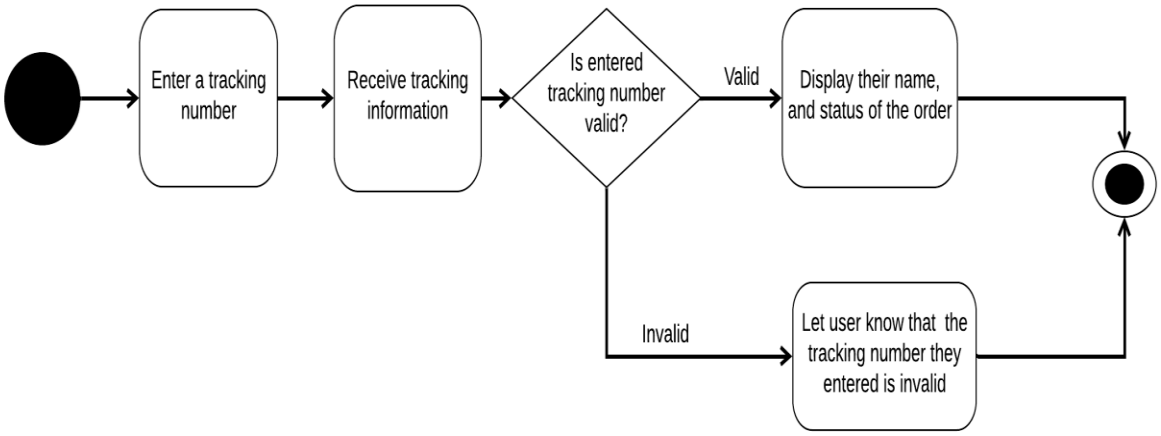
Activity Diagram: Track Order

Perspective: User



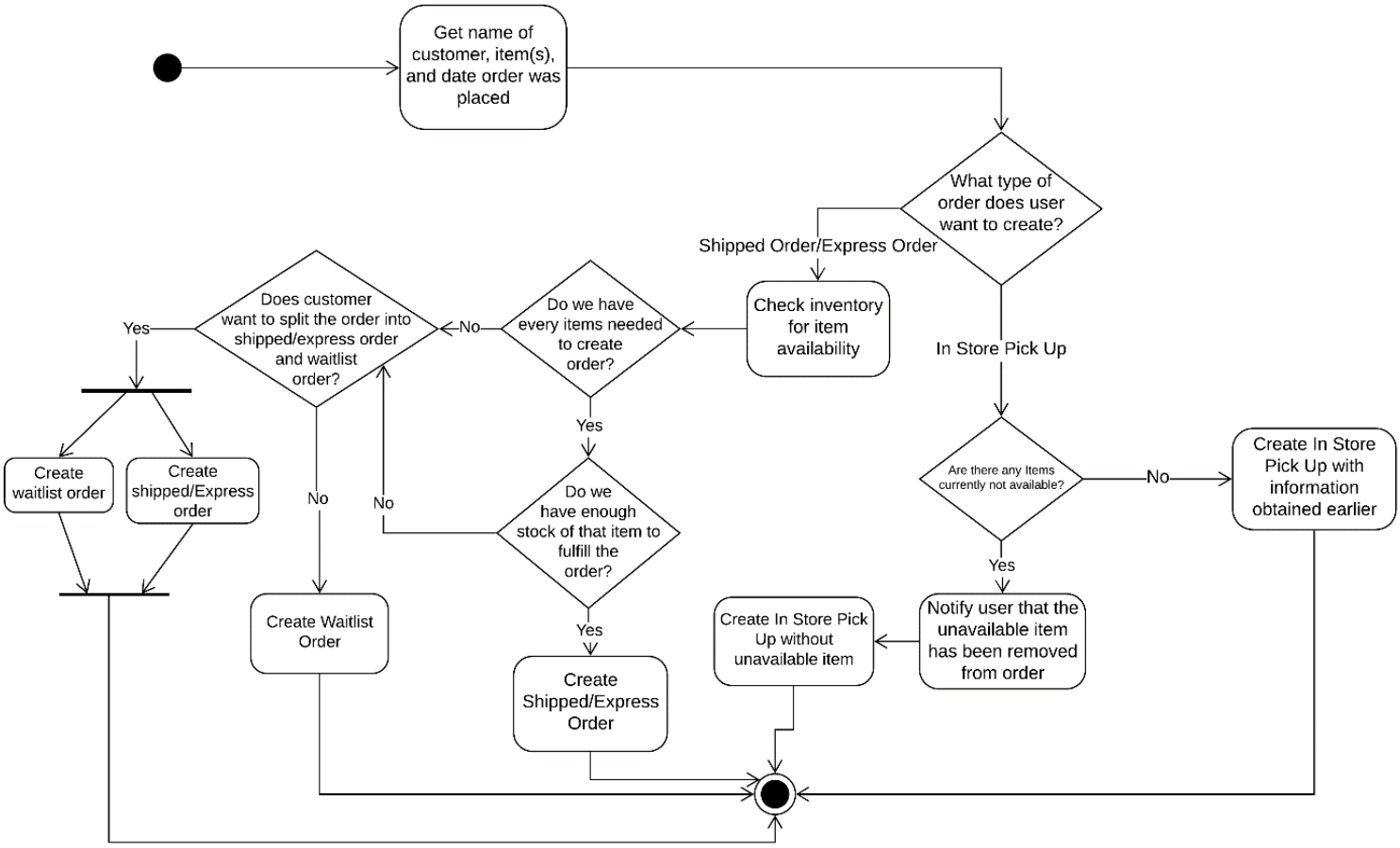
Activity Diagram: Alert Restock Supplies

Perspective: Company



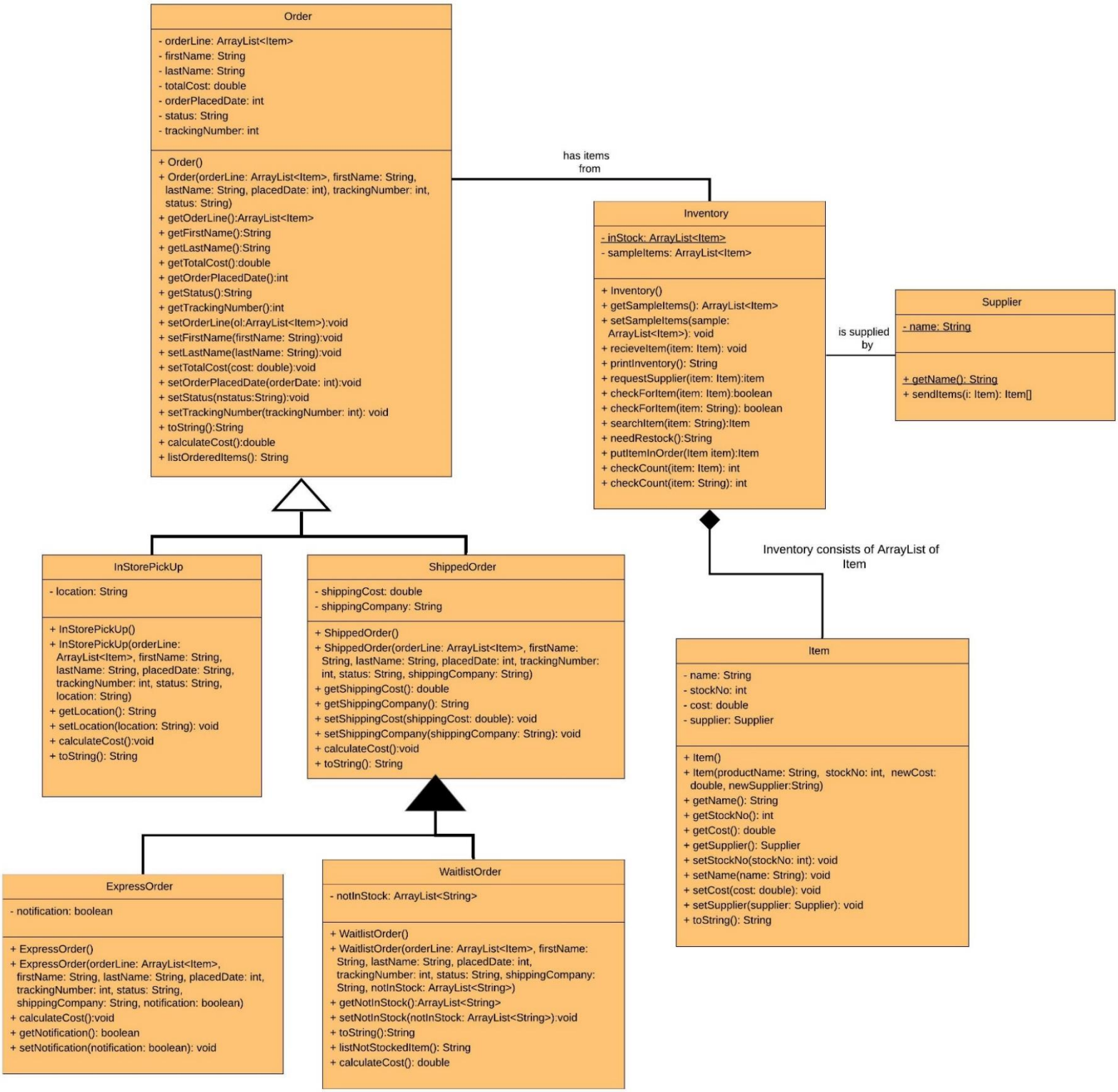
Activity Diagram: Process Order

Perspective: Company



Description of process order: Once we get information from the user to create the order, we will have separated cases depending on the type of order: in store pick up and shipped order/express order. After the type of order is determined, both cases check for stock availability. While shipped order and express order will either split the order into regular shipped/express order and waitlist order, in store pick up order will be created without items that are not in the stock.

UML Class Diagrams



The UML abstraction shows the methods/data members within classes and relationships between classes. Specifically, we have a hierarchy of Order classes where the parent class Order is an abstract class. The Inventory class has an aggregate relationship with the Item class and associations with both the Supplier and Order classes.

Implementation

How did we code?

Using a hierarchy of Order classes allowed us to distinguish the different types of orders and their functionality. This allowed us to have consistency in data members and methods with related classes demonstrating an “is a” relationship. This also allowed us to effectively follow the behavioral contract and have unique orders given the criteria of a given order. Orders were loaded into the system using an input file and then processed as a type of order (Shipped, Express, Waitlist, In-Store pickup) based on the contents of the Inventory. Creating an Inventory class provided us with a class that could dynamically hold all the items needed to fill orders. The Inventory has two ArrayLists. A static ArrayList is initially loaded through an input file representing the items in the inventory and an instance ArrayList which contains a sample of each Item for when they need to be restocked. The Driver then contains a static Inventory that is updated throughout the program. The functionality of the Inventory class also allows us to check how many of a given item are in an order and whether they need to alert the Supplier class for a restock. The Supplier class that was associated with the Inventory allows for us to “restock” the Inventory based on the needs of the system When alerted, the Supplier class method to restock returned 3 of the needed items.

The driver used various methods to load the inventory, create orders, allow for tracking of shipped orders, and issuing a statement of the orders created. The driver read in an input file to load the inventory and an input file of the orders to be created. After the orders are processed, there are two output files, one for the user and one for the company to keep track of the activity. This allows our program to accommodate to both the user and the owner. We used an integer to

keep track of the days and used a modulo operator to call for restock every 4 days. This allows for Waitlisted orders to get filled as the program runs.

Who did what?

We split up the classes evenly between group members: Hiro was tasked with the WaitListOrder and Item classes, Miriam completed InStorePickUp and ExpressOrder classes, Deandra coded the Order and ShippedOrder classes, and Jenny did the Inventory and Supplier classes. For the implementation of the Driver, Hiro and Miriam primarily coded the driver. Sample test suits in the report were coded by Hiro and Deandra. The formatting and putting together of project deliverables were done by Deandra and Jenny.

Test Cases

We have test cases for three classes: Inventory class, Order class and WaitlistOrder class. We decided to test Inventory class because our system relies heavily on the inventory and interact with them often as we create order. Decisions of what type of order should be created significantly depend on inventory methods so we wanted to make sure those methods do what they are supposed to do. We also picked Order class because it is the super class of all the order types. We wanted to make sure that the class is instantiated correctly and that helper methods in the driver class satisfy what the instance method in the class satisfies – the requirements for the class. We also tested WaitlistOrder class since it is different from any other order types and it is one of the most crucial and the most complex aspect of our system.

Reflection

Modifications of previous work (diagrams, classes, test cases, etc.) were needed as we progressed through the project but were much more limited with a planned approach. Furthermore, our use of a Class diagram, Use Case diagram, and Activity diagrams very much helped us throughout the process. While we did go back and make some adjustments as we started to code, we had a strong structure to use throughout the process. This showed us the importance of planning before diving into writing code.

We also learned that many different ideas could solve the same problem. Being able to collaborate on this project in a small group setting allowed all of us to share ideas we had to structure and implement various parts of the program. As our work progressed, and we checked in with each other, it was clear during the brainstorming progress that we all conceptualized solutions to certain problems in different ways. Specifically, coding instance methods for various related classes could be implemented in different ways. In scenarios like this, communication is key in making cohesive, functional code.

Thinking about code from a user's perspective is a great way to identify which elements of code should create output and which should be handled by the program itself. We learned this from re-creating certain Activity diagrams from the user's perspective rather than the programmer's perspective. This helped us decide which methods in the driver class should use an input file rather than asking for user input. The use of test cases was very helpful in compiling code from different programmers. When we went to compile all of the code we had written before implementing a driver class, going through the test cases for each class showed us

potential errors and disagreements between classes. This showed us exactly what we needed to fix without having to re-run and break down entire programs.

If we could go back and make changes, we would have outlined our structure for the driver class sooner. This would have allowed us to structure the input parameters of various instance methods for better functionality. In addition, this would have made the coding of the driver more efficient as we would not have had to go back and make changes in various classes. As we saw throughout the process, no part of planning ahead goes exactly as intended, but having a strong general idea of what is next can save having to re-do unnecessary mistakes. While we were able to use this strategy in many aspects of the project, we ultimately could have utilized this more when planning for the driver.

Code

Classes

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
public class Supplier {
	private static String name;
	public Supplier(String n) {
		name = n;
	}
	public static String getName() {
		return name;
	}
	public Item[] sendItems(Item i) {
		Item[] returning = new Item[3];
		for(int j = 0; j < 3; j++) {
			returning[j] = i;
		}
		return returning;
	}
}
```

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
import java.util.ArrayList;
public abstract class Order {
	// *** DATA MEMBERS *** //
	private ArrayList<Item> orderLine;
	private String firstName;
	private String lastName ;
	private double totalCost;
	private int trackingNumber;
	private String status;
	private int orderPlacedDate;
	// ---------------------------------------- //
	// *** CONSTRUCTORS *** //
	public Order() {
		orderLine = new ArrayList<Item>();
	}
	public Order(ArrayList<Item> oL, String fN, String lN, int placedDate, int trackNum, String status) {
		orderLine = oL;
		firstName = fN;
		lastName = lN;
		orderPlacedDate = placedDate;
		trackingNumber = trackNum;
		this.status = status;
	}
	// ---------------------------------------- //
	// *** GETTERS AND SETTERS *** //
	public ArrayList<Item> getOrderLine() {
		return orderLine;
```

```
}
public void setOrderLine(ArrayList<Item> orderLine) {
        this.orderLine = orderLine;
}
public String getFirstName() {
        return firstName;
}
public void setFirstName(String firstName) {
        this.firstName = firstName;
}
public String getLastName() {
        return lastName;
}
public void setLastName(String lastName) {
        this.lastName = lastName;
}
public int getTrackingNumber() {
        return trackingNumber;
}
public void setTrackingNumber(int newTrackingNum) {
        this.trackingNumber = newTrackingNum;
}
public String getStatus() {
        return status;
}
public void setStatus(String newStatus) {
        this.status = newStatus;
}
public double getTotalCost() {
        return totalCost;
}
```

```
public void setTotalCost(double totalCost) {
		this.totalCost = totalCost;
}
public int getOrderPlacedDate() {
		return orderPlacedDate;
}
public void setOrderPlacedDate(int orderPlacedDate) {
		this.orderPlacedDate = orderPlacedDate;
}
// ---------------------------------------- //
// *** OTHER METHODS *** //
// To calculate the total cost of an order
public double calculateCost() {
		double total = 0;
		for(int i = 0; i < this.orderLine.size(); i++) {
			total = total + this.orderLine.get(i).getCost();
		}
		setTotalCost(total);
		return total;
}
// To create a list of name of items in the order
public String listOrderedItems() {
		StringBuilder sb = new StringBuilder();
		String items;
		int currentIndex = 0;
		while(currentIndex < orderLine.size()) {
			String temp = orderLine.get(currentIndex).getName();
			sb.append(temp);
			if(currentIndex != orderLine.size() - 1) {
				sb.append(", ");
			}
```

```
                currentIndex++;
            }
            items = sb.toString();
            return items;
        }
        // Return a String containing information about an order
        public String toString() {
            String cost = String.format("%.2f", totalCost);
            return "Name: " + this.firstName + " " + this.lastName + "\nOrder Status: " + status +"\nOrder placed
            on day " + this.orderPlacedDate + "\nTracking number: "+trackingNumber+
            "\nOrder includes:" + listOrderedItems() + "\nTotal cost:$" + cost;
        }
}
```

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
import java.util.ArrayList;
public class ShippedOrder extends Order {
	// *** DATA MEMBERS *** //
	private double shippingCost;
	private String shippingCompany;
	// ---------------------------------------- //
	// *** CONSTRUCTORS *** //
	public ShippedOrder() {
		super();
	}
	public ShippedOrder(ArrayList<Item> oL, String fN, String lN, int placedDate, int trackingNumber, String status,
	String sCompany) {
		super(oL, fN, lN, placedDate, trackingNumber, status);
		shippingCompany = sCompany;
	}
	// ---------------------------------------- //
	// *** GETTERS AND SETTERS *** //
	public double getShippingCost() {
		return shippingCost;
	}
	public void setShippingCost(double shippingCost) {
		this.shippingCost = shippingCost;
	}
	public String getShippingCompany() {
		return shippingCompany;
	}
```

```
	public void setShippingCompany(String shippingCompany) {
		this.shippingCompany = shippingCompany;
	}
	// ---------------------------------------- //
	// *** OTHER METHODS *** //
	/* To calculate the cost of a shipped order
	* method should use the calculateCost() in the
	* super class and add shipping cost - 10% of the
	* total cost of the order
	*/
	public double calculateCost() {
		double finalCost = super.calculateCost();
		double sCost = finalCost * .10;
		setShippingCost(sCost);
		finalCost = finalCost + sCost;
		setTotalCost(finalCost);
		return finalCost;
	}
	/* Method returns a String containing information about a shipped order
	* including the shipping cost, shipping company and tracking
	* number
	*/
	public String toString() {
		String scost = String.format("%.2f", shippingCost);
		return "Shipped " + super.toString() + " (includes $" + scost + " shipping fee)\nShipped by " +
		this.shippingCompany;
	}
}
```

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
import java.util.ArrayList;
public class InStorePickUp extends Order{
	//data members
	private String location;
	//constructors
	public InStorePickUp() {
	}
	public InStorePickUp(ArrayList<Item> orderLine, String firstName, String lastName, int orderPlacedDate, int trackingNumber, String status, String location) {
		super(orderLine, firstName, lastName, orderPlacedDate, trackingNumber, status);
		this.location = location;
	}
	//getters and setters
	public String getLocation() {
		return location;
	}
	public void setLocation(String location) {
		this.location = location;
	}
	//other methods
	public String toString() {
		String s = "In Store Pickup " + super.toString() + "\nPickup location: " + location;
		return s;
	}
}
```

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
import java.util.ArrayList;
public class ExpressOrder extends ShippedOrder{
	//data members
	private boolean notification;
	//constructors
	public ExpressOrder() {
	}
	public ExpressOrder(ArrayList<Item> orderLine, String firstName, String lastName, int orderPlacedDate, int trackingNumber, String status, String shippingCompany, boolean notification) {
		super(orderLine, firstName, lastName, orderPlacedDate, trackingNumber, status, shippingCompany);
		this.notification = notification;
	}
	//getters and setters
	public boolean getNotification() {
		return notification;
	}
	public void setNotification(boolean notification) {
		this.notification = notification;
	}
	//other methods
	public double calculateCost() {
		double sum = super.calculateCost() + 4.00;
		setTotalCost(sum);
		return sum;
	}
	public String toString() {
		String cost = String.format("%.2f", getTotalCost());
		String scost = String.format("%.2f", getShippingCost());
```

```
            String s = super.toString() + "\nExpress Fee: A $4.00 Express Fee has already been added to the Total
            Cost.";

            return "Express Order for " + getFirstName() + " " + getLastName() + "\nOrder Status: " + getStatus() +
            "\nTracking Number: " + getTrackingNumber() +  "\nOrder placed on day " + getOrderPlacedDate() +

            "\nOrder includes: " + listOrderedItems() + "\nTotal cost: $" + cost + " (includes $" + scost + " shipping
            fee and $4.00 express fee)\nShipped by " + getShippingCompany() + "\nNotifications?: " + notification;

        }
}
```

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
import java.util.*;
public class WaitlistOrder extends ShippedOrder{
	//data members----------------------------------------------
	private ArrayList<String> notInStock;
	//-------------------------------------------------------
	//constructors
	public WaitlistOrder() {
		super();
		notInStock = new ArrayList<String>();
	}
	public WaitlistOrder(ArrayList<Item> orderline,String firstName, String lastName, int placedDate, int
	trackingNumber, String status, String shippingCompany, ArrayList<String> notInStock) {
		super(orderline,firstName,lastName,placedDate, trackingNumber, status, shippingCompany);
		setShippingCost(0);
		this.notInStock = notInStock;
	}
	//-------------------------------------------------------
	//Getters and setters----------------------------------------
	public ArrayList<String> getNotInStock(){
		return notInStock;
	}
	public void setNotInStock(ArrayList<String> notInStock) {
		this.notInStock = notInStock;
	}
	//-------------------------------------------------------
	public double calculateCost() {
		double total = 0;
		if(super.getOrderLine().isEmpty()) {
```

```
                return total;
        }
        else {
                for(int i = 0;i<super.getOrderLine().size();i++) {
                        double cost = super.getOrderLine().get(i).getCost();
                        total = total +cost;
                }
        }
        setTotalCost(total);
        return total;
}
//toString method: this calls the toString method in super class and add expected ship date and list of name of the item needed for this order to be complete.
public String toString() {
        String cost = String.format("%.2f", getTotalCost());
        return "Waitlist Order for " + getFirstName() + " " + getLastName() + "\nOrder Status: " + getStatus() + "\nTracking Number: " + getTrackingNumber() +  "\nOrder placed on day " + getOrderPlacedDate() +
        "\nOrder includes: " + listOrderedItems() + "\nTotal cost:$" + cost + "\nItems needed to complete order: " + listNotStockedItem();
}
//listNotStockedItem method: this method is going to traverse the ArrayList of String that contains list of name of item,
//if this order is complete and didn't have any names, String will be N/A,
//otherwise We create a String of a list of name of items by using StringBuilder
public String listNotStockedItem() {
        String itemlisted;
        StringBuilder sb = new StringBuilder();
        if(notInStock.isEmpty()) {
                itemlisted = "N/A";
        }
        else {
                int currentIndex = 0;
                while(currentIndex < notInStock.size()-1){
```

```
                    String temp = notInStock.get(currentIndex);
                    sb.append(temp);
                    sb.append(", ");
                    currentIndex++;
                }
                String last = notInStock.get(currentIndex);
                sb.append(last);
                itemlisted = sb.toString();
            }
            return itemlisted;
        }
}
```

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
import java.util.*;
public class Inventory {
	private static ArrayList<Item> inStock;
	private ArrayList<Item> SampleItems;
	//-------------------------------------------------------
	public Inventory() {
		inStock = new ArrayList<>();
		SampleItems = new ArrayList<>();
	}
	//-------------------------------------------------------
	public ArrayList<Item> getSampleItems(){
		return SampleItems;
	}
	public void setSampleItems(ArrayList<Item> sample) {
		SampleItems = sample;
	}
	//Takes an item as input and adds it to the inventory
	public void recieveItem(Item i) {
		inStock.add(i);
	}
	//-------------------------------------------------------
	//Checks to see if an item is in the Inventory
	//Does not remove the item
	//returns true if present, false if not
	public boolean checkForItem(Item item) {
		for(int i = 0; i < inStock.size(); i++) {
			if(item.getStockNo()==inStock.get(i).getStockNo()) {
```

```
                    return true;
                }
            }
            return false;
    }
    //-------------------------------------------------------
    // overloaded method
    public boolean checkForItem(String item) {
            for(int i = 0; i < inStock.size(); i++) {
                if(item.equals(inStock.get(i).getName()) ){
                    return true;
                }
            }
            return false;
    }
    //-------------------------------------------------------
    public Item searchItem(String item) {
    int position = 0;
    for(int i = 0;i< inStock.size();i++) {
            if(item.equals(inStock.get(i).getName())) {
                position = i;
                Item target = inStock.get(position);
                return target;
            }
    }
    Item fail = new Item();
    fail.setStockNo(-1);
    return fail;
    }
    //-------------------------------------------------------
    //Goes through the inventory and adds the item needed to the order
```

```
//Additionally removes that item from the inventory
public Item putItemInOrder(Item item) {
		for(int i = 0; i < inStock.size(); i++) {
			if(item.getStockNo()== inStock.get(i).getStockNo()) {
				Item temp = inStock.get(i);
				inStock.remove(i);
				return temp;
			}
		}
		System.out.println("Item not found.");
		Item bad = new Item();
		bad.setStockNo(-1);
		return bad;
}
//---------------------------------------------------------
//Takes an item and sees how many of that item are in the inventory
public int checkCount(Item item) {
int count = 0;
for(int i = 0; i < inStock.size(); i++) {
		if( item.getStockNo()== inStock.get(i).getStockNo()) {
			count++;
		}
}
return count;
}
//---------------------------------------------------------
//overloading method of checkCount
public int checkCount(String item) {
int count = 0;
for(int i = 0; i < inStock.size(); i++) {
		if( item.equals(inStock.get(i).getName())) {
```

```
                count++;
            }
        }
        return count;
    }
    //-----------------------------------------------------
    //Checks how many of each item are in the inventory
    //If the number of a particular item is less than 3
    //Adds the name of the item to a String that is returned
    //The String is the names of all the items 3 or less in the inventory
    //Separated by "&"
    public String needRestock() {
    //Create a StringBuilder that starts with the & character
    StringBuilder sb = new StringBuilder();
    sb.append("&");
    //Make an arraylist to store the names of checked items
    ArrayList<String> names = new ArrayList<>();
    //Loop through the array of items
    for(int i = 0; i < inStock.size(); i++) {
            //If there are less than 3 of the item in stock, make sure
            //We havent checked the item then add it to the list
            if(checkCount(inStock.get(i)) < 3) {
                boolean checked = false;
                int k = 0;
                while(checked == false && k < names.size()) {
                    if(names.get(k).equals(inStock.get(i).getName())) {
                        checked = true;
                    }
                    k++;
                }
                //If we haven't checked it and it's less than 3
```

```
                        //Add it to the list of needed Strings
                        if(checked == false) {
                                sb.append(inStock.get(i).getName() + "&");
                                names.add(inStock.get(i).getName());
                        }
                }
        }
        if(sb.toString().equals("&")) {
                return "No items needed to be restocked";
        }
        return sb.toString();
        }
        //Method that identifies the name of the item and returns it
        //To be told to the supplier
        //Takes and returns the  item
        public Item requestItem(Item name) {
                System.out.println("We request 3 more " + name.getName());
                return name;
        }
        //-------------------------------------------------------
        public String printInventory() {
                StringBuilder value = new StringBuilder();
                for(int k = 0; k < inStock.size() - 1; k++) {
                        value.append(inStock.get(k).getName() + "\n");
                }
                if(inStock.size() > 0) {
                        value.append(inStock.get(inStock.size() - 1).getName());
                }
                return value.toString();
        }
}
```

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
public class Item {
	//Data members
	private String name;
	private int stockNo;
	private double cost;
	private Supplier supplier;
	//---------------------------------------------------------------
	//constructors----------------------------------------------------
	public Item() {
	}
	public Item(String productName,int stockNo,double newCost,Supplier newSupplier) {
		name = productName;
		this.stockNo = stockNo;
		cost = newCost;
		supplier = newSupplier;
	}
	//getters---------------------------------------------------------
	public String getName() {
		return name;
	}
	public int getStockNo() {
		return stockNo;
	}
	public double getCost() {
		return cost;
	}
	public Supplier getSupplier() {
```

```
		return supplier;
	}
	//setters------------------------------------------------------------
	public void setName(String newPName) {
		name = newPName;
	}
	public void setStockNo(int newStock) {
		stockNo = newStock;
	}
	public void setCost(double newCost) {
		cost = newCost;
	}
	public void setSupplier(Supplier newSupplier) {
		supplier = newSupplier;
	}
	//toString method
	public String toString() {
		return "Product name: " + name + "\nStock number: "+stockNo + "\nCost: "+ cost
		+ "\nSupplier: " + Supplier.getName();
	}
}
```

Test Suits

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
class InventoryTest {
	//-------------------------------------------------------
	@Test
	void testRecieveItem() {
		Supplier s = new Supplier("US");
		Item item = new Item("Paper towel", 1 , 3.00 , s);
		Inventory i = new Inventory();
		i.recieveItem(item);
		int num = i.checkCount(item);
		assertTrue(num == 1);
	}
	//-------------------------------------------------------
	@Test
	void testConstructor() {
		Inventory i = new Inventory();
		assertNotNull(i);
	}
	//-------------------------------------------------------

	@Test
	void testCheckForItemInInventory() {
		Supplier s = new Supplier("US");
		Item item = new Item("Paper towel", 1 , 3.00 , s);
		Inventory i = new Inventory();
```

```
        i.recieveItem(item);
        boolean value = i.checkForItem("Paper towel");
        assertTrue(value == true);
}
//-------------------------------------------------
@Test
void testCheckForItemNotInInventory() {
        Inventory i = new Inventory();
        boolean value = i.checkForItem("Towel");
        assertTrue(value == false);
}
//-------------------------------------------------
@Test
void testPutItemInOrder() {
        Supplier s = new Supplier("US");
        Item item = new Item("Paper towel", 1 , 3.00 , s);
        Inventory i = new Inventory();
        i.recieveItem(item);
        Item value = i.putItemInOrder(item);
        assertTrue(value.getStockNo() == item.getStockNo());
}
//-------------------------------------------------
@Test
void testPutItemInOrderNotInInventory() {
        Supplier s = new Supplier("US");
        Item item = new Item("Paper towel", 1 , 3.00 , s);
        Inventory i = new Inventory();
        Item value = i.putItemInOrder(item);
        assertTrue(value.getStockNo() == -1);
}
//-------------------------------------------------
```

```
@Test
void testSearchItemInInventory() {
		Inventory i = new Inventory();
		Supplier s = new Supplier("US");
		Item temp = new Item("Paper towel", 1 , 3.00 , s);
		i.recieveItem(temp);;
		Item temp2 = i.searchItem("Paper towel");
		assertEquals(temp, temp2);
}
//-------------------------------------------------------
@Test
void testSearchItemNotInInventory() {

		Inventory i = new Inventory();
		Supplier s = new Supplier("US");
		Item temp = new Item("Paper towel", 1 , 3.00 , s);
		i.recieveItem(temp);;
		Item temp2 = i.searchItem("Towel");
		assertFalse(temp.equals(temp2));

}
//-------------------------------------------------------
@Test
void testCheckCount() {
		Supplier s = new Supplier("US");
		Item item = new Item("Paper towel", 1 , 3.00 , s);
		Item item2 = new Item("Paper towel", 1 , 3.00 , s);
		Inventory i = new Inventory();
		i.recieveItem(item);
		i.recieveItem(item2);
		Item paperTowel = new Item("Paper towel", 1 , 3.00 , s);
```

```
        int count = i.checkCount(paperTowel);
        assertTrue(count == 2);
}
//-------------------------------------------------------
@Test
void testCheckCountZero() {
        Supplier s = new Supplier("US");
        Inventory i = new Inventory();
        Item paperTowel = new Item("Paper towel", 1 , 3.00 , s);
        int count = i.checkCount(paperTowel);
        assertTrue(count == 0);
}
//-------------------------------------------------------
@Test
void testprintInventory() {
        Supplier s = new Supplier("US");
        Item item = new Item("Paper towel", 1 , 3.00 , s);
        Item item2 = new Item("Lemon", 2 , 3.00 , s);
        Item item3 = new Item("Lemon", 2 , 3.00 , s);
        Inventory i = new Inventory();
        i.recieveItem(item);
        i.recieveItem(item2);
        i.recieveItem(item3);
        String value = i.printInventory();
        String desired = "Paper towel" + "\n" + "Lemon" + "\n" + "Lemon";
        assertTrue(desired.equals(value));
}
//-------------------------------------------------------
@Test
void testprintInventoryEmpty() {
        Inventory i = new Inventory();
```

```
		String value = i.printInventory();
		String desired = "";
		assertTrue(desired.equals(value));
}
//-----------------------------------------------------
@Test
void testRequestItem() {
		Inventory i = new Inventory();
		Supplier s = new Supplier("s");
		Item item = new Item("Paper towel", 1 , 3.00 , s);
		Item found = i.requestItem(item);
		assertTrue(found.equals(item));
}
//-----------------------------------------------------
@Test
void testNeedRestock() {
		Supplier s = new Supplier("US");
		Item item = new Item("Paper towel", 1 , 3.00 , s);
		Item item2 = new Item("Lemon", 2 , 3.00 , s);
		Item item3 = new Item("Lemon", 2 , 3.00 , s);
		Inventory i = new Inventory();
		i.recieveItem(item);
		i.recieveItem(item2);
		i.recieveItem(item3);
		String value = i.needRestock();
		String desired = "&Paper towel&Lemon&";
		assertTrue(value.equals(desired));
}
//-----------------------------------------------------
@Test
void testNeedRestockNONE() {
```

```
		Supplier s = new Supplier("US");
		Item item = new Item("Paper towel", 1 , 3.00 , s);
		Inventory i = new Inventory();
		i.recieveItem(item);
		i.recieveItem(item);
		i.recieveItem(item);
		String value = i.needRestock();
		String desired = "No items needed to be restocked";
		assertTrue(value.equals(desired));
	}
	//-----------------------------------------------------
	@Test
	void testPrintInventory() {
		Supplier s = new Supplier("US");
		Item item = new Item("Paper towel", 1 , 3.00 , s);
		Item item2 = new Item("Lemon", 2 , 3.00 , s);
		Inventory i = new Inventory();
		i.recieveItem(item);
		i.recieveItem(item2);
		String list = i.printInventory();
		String expected = "Paper towel\nLemon";
		assertEquals(expected, list);
	}
}
```

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import java.util.*;
class OrderTest {
	@Test
		void testOrderNoArgs() {
		Order o = new ShippedOrder(); // Order class cannot be instantiated,
						// only if the actual type is one of its subclass
		assertTrue(o instanceof Order);
	}
	// ---------------------------------------- //
	@Test
	void testOrderArgs() {
		ArrayList<Item> orderLine = new ArrayList<>();
		Supplier s = new Supplier("Spring");
		Item a = new Item("Basketball", 1, 34.00, s);
		orderLine.add(a);
		Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
		assertTrue(o instanceof Order);
	}
	// ---------------------------------------- //
	@Test
	void testGetOrderLine() {
		ArrayList<Item> orderLine = new ArrayList<>();
		Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
		ArrayList<Item> actual = o.getOrderLine();
		assertEquals(orderLine, actual);
```

```
}
// ---------------------------------------- //
@Test
void testSetOrderLine() {
		ArrayList<Item> orderLine = new ArrayList<>();
		ArrayList<Item> expected = new ArrayList<>();
		Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
		Supplier s = new Supplier("FedEx");
		Item a = new Item("Roku", 1, 134.00, s);
		 orderLine.add(a);
		 expected.add(a);

		assertEquals(o.getOrderLine(), expected);
}
// ---------------------------------------- //
@Test
void testGetFirstName() {
		ArrayList<Item> orderLine = new ArrayList<>();
		Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
		String fname = o.getFirstName();
		assertTrue(fname.equals("Deandra"));
}
// ---------------------------------------- //
@Test
void testSetFirstName() {
		ArrayList<Item> orderLine = new ArrayList<>();
		Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
		o.setFirstName("Miriam");
		assertTrue(o.getFirstName().equals("Miriam"));
}
// ---------------------------------------- //
```

```
@Test
void testGetLastName() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        String lname = o.getLastName();
        assertTrue(lname.equals("Martin"));
}
// ---------------------------------------- //
@Test
void testSetLastName() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        o.setLastName("Scheinblum");
        assertTrue(o.getLastName().equals("Scheinblum"));
}
// ---------------------------------------- //
@Test
void testGetTrackingNumber() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        int trackNo = o.getTrackingNumber();
        assertEquals(4, trackNo);
}
// ---------------------------------------- //
@Test
void testSetTrackingNumber() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        o.setTrackingNumber(6);
        assertEquals(6, o.getTrackingNumber());
}
```

```
// ---------------------------------------- //
@Test
void testGetStatus() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        String status = o.getStatus();
        assertTrue(status.equals("Shipped"));
}
// ---------------------------------------- //
@Test
void testSetStatus() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        o.setStatus("Processing Order");
        assertTrue(o.getStatus().equals("Processing Order"));
}
// ---------------------------------------- //
@Test
void testGetTotalCostIfOrderLineIsEmpty() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        double totalCost = o.getTotalCost();
        assertTrue(totalCost == 0.0);
}
// ---------------------------------------- //
@Test
void testCalculateCost() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        Supplier s = new Supplier("Amazon");
        Item a = new Item("Roku", 1, 134.00, s);
```

```
        Item b = new Item("Emerson TV", 1, 500.50, s);
        orderLine.add(a);
        orderLine.add(b);
        double cost = o.calculateCost();
        assertEquals(o.getTotalCost(), cost);
}
// ---------------------------------------- //
@Test
        void testGetTotalCostIfOrderLineIsNotEmpty() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        Supplier s = new Supplier("Amazon");
        Item a = new Item("Roku", 1, 134.00, s);
        Item b = new Item("Emerson TV", 1, 500.50, s);
        orderLine.add(a);
        orderLine.add(b);
        double totalCost = o.calculateCost();
        double cost = o.getTotalCost();
        assertEquals(cost, totalCost);
}
// ---------------------------------------- //
@Test
void testSetTotalCost() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        o.setTotalCost(100.00);
        assertTrue(o.getTotalCost() == 100.00);
}
// ---------------------------------------- //
@Test
void testGetOrderPlacedDate() {
```

```
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        int date = o.getOrderPlacedDate();
        assertEquals(4, date);
}
// ---------------------------------------- //
@Test
void testSetOrderPlacedDate() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        o.setOrderPlacedDate(8);
        assertEquals(8, o.getOrderPlacedDate());
}
// ---------------------------------------- //
@Test
void testListOrderedItems() {
        ArrayList<Item> orderLine = new ArrayList<>();
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");
        Supplier s = new Supplier("Amazon");
        Item a = new Item("Roku", 1, 134.00, s);
        Item b = new Item("Emerson TV", 1, 500.50, s);
        orderLine.add(a);
        orderLine.add(b);
        String list = o.listOrderedItems();
        String expected = "Roku,Emerson TV";
        assertEquals(expected, list);
}
// ---------------------------------------- //
@Test
void testToString() {
        ArrayList<Item> orderLine = new ArrayList<>();
```

```
        Order o = new ShippedOrder(orderLine, "Deandra", "Martin", 4, 4, "Shipped", "Spring");

        Supplier s = new Supplier("Amazon");

        Item a = new Item("Roku", 1, 134.00, s);

        Item b = new Item("Emerson TV", 1, 500.50, s);

        orderLine.add(a);

        orderLine.add(b);

        String actual = o.toString();

        String expected = "Shipped Order\nName: Deandra Martin\nOrder placed on day 4\nTracking
        number: 4\nOrder includes:Roku,Emerson TV\n"

        + "Total cost:$0.00 (includes $0.00 shipping fee)\nShipped by Spring";

        assertEquals(expected, actual);

    }

}
```

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum
// Group Project
//CMS270
// April 24 2020
import static org.junit.jupiter.api.Assertions.*;
import java.util.*;
import java.util.Arrays;
import org.junit.jupiter.api.Test;
class WaitlistOrderTest {
	// test for the non-arg constructor of the the waitlist class
	@Test
	void testWaitlistOrderNon() {
		WaitlistOrder test = new WaitlistOrder();
		assertTrue(test instanceof WaitlistOrder);
	}
	// testing for the waitlist constructor with list of parameters.
	@Test
	void testWaitlistOrderArgs() {
		Item a = new Item("Basketball",1,34.00,new Supplier("Spring"));
		Item b = new Item("Nike Kobe",2,123.00,new Supplier("Nike"));
		ArrayList<Item> orderline = new ArrayList<Item>(Arrays.asList(a,b));
		ArrayList<String> notInStock = new ArrayList<String>(Arrays.asList("Basketball Rim", "Book"));
		String firstName = "Hiroki";
		String lastName = "Sato";
		int placedDate = 1;
		String status = "Waitlisted";
		String shippingCompany = "Amazon";
		int trackingNumber = 1;
```

```
        WaitlistOrder expected = new
        WaitlistOrder(orderline,firstName,lastName,placedDate,trackingNumber,status,shippingCompany,notI
        nStock);
        assertTrue(expected instanceof WaitlistOrder);
}
//testing if NotInStock, ArrayList of String are returned properly
@Test
void testGetNotInStock() {
        Item a = new Item("Basketball",1,34.00,new Supplier("Spring"));
        Item b = new Item("Nike Kobe",2,123.00,new Supplier("Nike"));
        ArrayList<Item> orderline = new ArrayList<Item>(Arrays.asList(a,b));
        ArrayList<String> expected = new ArrayList<String>(Arrays.asList("LED Lightbulb", "4K Cannon
        Camera"));
        String firstName = "Hiroki";
        String lastName = "Sato";
        int placedDate = 1;
        String status = "Waitlisted";
        String shippingCompany = "Amazon";
        int trackingNumber = 1;
        WaitlistOrder test = new
        WaitlistOrder(orderline,firstName,lastName,placedDate,trackingNumber,status,shippingCompany,exp
        ected);
        ArrayList<String> actual = test.getNotInStock();
        assertEquals(expected,actual);
}
// testing if setNotInStock will set the ArrayList of String properly
@Test
void testSetNotInStock() {
```

```
        ArrayList<String> test = new ArrayList<String>(Arrays.asList("Toothbrush", "4K Cannon Camera"));
        WaitlistOrder testOrder = new WaitlistOrder();
        testOrder.setNotInStock(test);
        // if we were able to successfully set the ArrayList of String, what getter returns should not be empty.
        assertFalse(testOrder.getNotInStock().isEmpty());
}
//when we did have names of item that needed for the order, this method test if listNotStcokedItem
// will create the string of name of items needed for the order.
@Test
void testListNotStockedItemNotEmpty() {
        String item1 = "LED";
        String item2 = "4K Cannon Camera";
        String item3 = "Toshiba TV";
        WaitlistOrder test = new WaitlistOrder();
        String expected = item1 +", "+item2 + ", "+item3;
        ArrayList<String> temp = new ArrayList<String>(Arrays.asList(item1,item2,item3));
        test.setNotInStock(temp);
        String actual = test.listNotStockedItem();
        assertTrue(expected.equals(actual));
}
// if we don't have any items needed to complete the order, the list should return N/A
// testing to see if it returns the correct value.
@Test
void testListNotStockedItemEmpty() {
        WaitlistOrder test = new WaitlistOrder();
        String expected = "N/A";
        String actual = test.listNotStockedItem();
```

```
        assertEquals(expected,actual);
}
// calculateCost in waitlist order doesn't add shipping cost to the total cost,
// we are testing to see if the calculateCost returns the correct value
@Test
void testCalculateCost() {
        Item a = new Item("Basketball",1,34.00,new Supplier("Spring"));
        Item b = new Item("Nike Kobe",2,123.00,new Supplier("Nike"));
        ArrayList<Item> orderline = new ArrayList<Item>(Arrays.asList(a,b));
        // expected is the subtotal of the cost of all items in the waitlistOrder
        double expected = 0;
        for(int i = 0;i<orderline.size();i++) {
                double cost = orderline.get(i).getCost();
                expected = expected +cost;
        }
        WaitlistOrder test = new WaitlistOrder();
        test.setOrderLine(orderline);
        double actual = test.calculateCost();
        assertEquals(expected,actual);
}
// Testing if toString will return the value with expected format.
@Test
void testToString() {
        Item a = new Item("Basketball",1,34.00,new Supplier("Spring"));
        Item b = new Item("Nike Kobe",2,123.00,new Supplier("Nike"));
        ArrayList<Item> orderline = new ArrayList<Item>(Arrays.asList(a,b));
        ArrayList<String> notInStock = new ArrayList<String>(Arrays.asList("Basketball Rim", "Book"));
```

```
        String firstName = "Hiroki";
        String lastName = "Sato";
        double subTotal = a.getCost() + b.getCost();
        String total = String.format("%.2f", subTotal);
        int placedDate = 1;
        String shippingCompany = "Amazon";
        int trackingNumber = 1;
        String orderedItems = a.getName() + ", " +b.getName();
        WaitlistOrder expected = new
        WaitlistOrder(orderline,firstName,lastName,placedDate,trackingNumber,"waitlisted",shippingCompan
        y,notInStock);
        String superStr = "Waitlist Order for "+ firstName +" "+lastName +"\nOrder Status:
        waitlisted"+"\nTracking Number: " + trackingNumber
        + "\nOrder placed on day "+placedDate+"\nOrder includes: "+ orderedItems
        + "\nTotal cost:$"+ total;
        String expectedStr = superStr + "\nItems needed to complete order: "+ expected.listNotStockedItem();
        expected.calculateCost();
        String actual = expected.toString();
        assertEquals(expectedStr,actual);
    }
}
```

Driver

```
// Deandra Martin, Jenny Goldsher, Hiroki Sato, Miriam Scheinblum

// Group Project

//CMS270

// April 24 2020

import java.util.*;

import java.io.*;

public class RunSystem {

	private static Inventory inventory;

	private static int day;

	private static int trackingNumber;

	//-------------------------------------------------------

	public static void main(String[] args) {
		day=0;
		trackingNumber=1;
		Scanner userInput = null;
		PrintWriter printUpdate = null;
		PrintWriter printActivity = null;
		// all of our input/output files needed to take inputs and print output
		File file1 = new File("Inventory.txt");
		File file2 = new File("UserInput.txt");
		File customerOutput = new File("CustomerOutput.txt");
		File companyActivity = new File("OrderLog.txt");
		String restockRequest = "N/A";
		ArrayList<Order> orderHolder = new ArrayList<Order>();
		// loading inventory based on the input file, Inventory.txt
		loadInventory(file1);
		try {
			//2. read input from a customer.
			userInput = new Scanner(file2);
			printUpdate = new PrintWriter(customerOutput);
			printActivity = new PrintWriter(companyActivity);
			//helper method 10;
			printWelcomeMessage(printUpdate);
```

```
while(userInput.hasNext()) {
        day+=1;
        notify(orderHolder, printUpdate);
        //every 4 days we check if there's any thing to be restocked and restock.
        if(day%4 == 0){
                //if restockRequest is not equal to "No items needed to be restocked."
                if(!(restockRequest.equals("No items needed to be restocked"))) {
                        restockInventory(restockRequest);
                }
        }
        for(int i = 0; i < orderHolder.size(); i++) {
                //express order status changes each day
                if (orderHolder.get(i) instanceof ExpressOrder) {
                        if (orderHolder.get(i).getStatus().equals("Order Placed")){
                                orderHolder.get(i).setStatus("Shipped");
                        }
                        else {
                                orderHolder.get(i).setStatus("Delivered");
                        }
                }
                else if(orderHolder.get(i) instanceof InStorePickUp) {
                        orderHolder.get(i).setStatus("Ready for pickup");
                }
                else if( orderHolder.get(i) instanceof WaitlistOrder) {
                        WaitlistOrder check = (WaitlistOrder)orderHolder.get(i);
                        if(!(check.listNotStockedItem().equals("N/A"))) {
                                check = WaitlistOrderUpdate(check);
                                if(check.getStatus().equals("Auto-Cancel")) {
                                //print out the name on the customer output and tell
                                them your waitlist order had been cancelled.
                                // then remove the order
                                printCancelMessage(check,printUpdate);
                                orderHolder.remove(i);
                                }
                        }
                }
                else{
                        // this represents how many days it has passed after order
                        was placed.
                        int time = day - orderHolder.get(i).getOrderPlacedDate();
                        if(time == 2) {
                                orderHolder.get(i).setStatus("Shipped");
                        }
```

```
                    if(time == 3 || time == 4) {
                        Random r = new Random();
                        int deliverSpeed = r.nextInt(1);
                        if (deliverSpeed == 0) {
                        orderHolder.get(i).setStatus("Delivered");
                        }
                    }
                    else if(time == 5) {
                        orderHolder.get(i).setStatus("Delivered");
                    }
                }
            }
            String action = userInput.nextLine();
            ArrayList<Order> tempOrder;
                if(action.equals("MO")) {
                    //call create order
                    String customerInfo = userInput.nextLine();
                    //read the next line that contains information of Item
                    String orderedItem = userInput.nextLine();
                    //String[]items = orderedItem.split(",");
                    tempOrder =
                createOrder(customerInfo,orderedItem,printUpdate);
                    if(tempOrder.get(0).getTrackingNumber()!= -1) {
                        for(int i = 0;i < tempOrder.size();i++) {
                            Order temp = tempOrder.get(i);
                                orderHolder.add(temp);
                        }
                    }
                }
                else if(action.equals("T")) {
                    //call trackOrder
                    String data = userInput.nextLine();
                    int trackingNumber = Integer.parseInt(data);
                    TrackOrder(orderHolder,printUpdate,trackingNumber);
                }
                restockRequest = inventory.needRestock();
                updateCompanyActivity(orderHolder, printActivity);
                //removes orders that have been delivered already before today
                for(int i = 0; i < orderHolder.size(); i++) {
                String status = orderHolder.get(i).getStatus();
                if (status.equals("Delivered")) {
                    //we should let the company know that this order had been
                    delivered and will be deleted from the systen.
```

```
                                        printWarningOrderDelivered(orderHolder.get(i),printActivity);
                                        orderHolder.remove(i);
                                }
                                else if (status.contentEquals("Ready for pickup")) {
                                        Random r = new Random();
                                        int n = r.nextInt(2);
                                        //if random integer is 1, the customer has picked up the order
                                        //otherwise, we leave the order in the system and check
                                        again tomorrow
                                        if (n == 1) {
                                                printWarningOrderPickedUp(orderHolder.get(i),
                                        printActivity);
                                                orderHolder.remove(i);
                                        }
                                }
                        }
                }
        }
        catch(FileNotFoundException e) {
                System.out.println("File could not be found.");
        }
        finally {
                if(userInput != null) {
                        userInput.close();
                }
                if(printUpdate != null) {
                        printUpdate.close();
                }
                if(printActivity != null) {
                        printActivity.close();
        }
}

public static void loadInventory(File file)  {
        inventory = new Inventory();
        ArrayList<Item>sample = new ArrayList<>();

        try {
                Scanner in = new Scanner(file);
                String supplierName = in.nextLine();
                while(in.hasNextLine()) {
                        String line = in.nextLine();
```

```
                String[] tokens = line.split(",");
                int quantity = Integer.parseInt(tokens[0]);
                String name = tokens[1];
                double cost = Double.parseDouble(tokens[3]);
                int sNum = Integer.parseInt(tokens[2]);
                Supplier s = new Supplier(supplierName);
                        for(int i = 0;i<quantity;i++) {
                                Item temp = new Item(name,sNum,cost,s);
                                inventory.recieveItem(temp);
                        }
                        Item Itemsample = new Item(name,sNum,cost,s);
                        sample.add(Itemsample);
                }
        inventory.setSampleItems(sample);
        }
        catch(FileNotFoundException e) {
                System.out.println("File could not be found.");
        }
}

// Read the line from scanner that we pas from the argument.
//determines what type of order should be created and returns the order
//this is possible by definition of inclusion polymorphism.
public static ArrayList<Order> createOrder(String customerInfo,String ordered,PrintWriter out)
{
        ArrayList<Order> order = new ArrayList<Order>();

        //read the line for customer information and broke the string into array of string
        String[] token = customerInfo.split(",");

        //read the next line that contains information of Item
        //String orderedItem = scan.nextLine();
        String[]items = ordered.split(",");
                if(token[0].equals("SO")) {
                        //call create shipped order.
                        order = createShippedOrder(token,items);
                        //also, for when item is not available take last index of array that
                        contains whether to merge order or split.
                }
                else if(token[0].equals("EXPO")) {
                        //call createExpress order
                        order = createExpressOrder(token,items);
```

```
                        //also, for when item is not available take last index of array that
                        contains whether to merge order or split
                }
                else if(token[0].equals("ISPU")) {
                        //call create InStorePickUp
                        InStorePickUp ISPU = createInStorePickUp(token,items,out);
                        order.add(ISPU);
                }
                return order;
        }

//---------------------------------------------------------------------------------------

//TrackOrder method: This is one of the functionality that user have access to, it takes the tracking
number
// that customer provided and search through the orderHolder, if the tracking number matches it
//it returns the name of customer and status of the order.
public static void TrackOrder(ArrayList<Order> orderHolder,PrintWriter printUpdate, int trackNum) {
        //first check if orderHolder was empty or not
        if(orderHolder.isEmpty() || trackNum > trackingNumber){
                printUpdate.println("-----------------------------------------------------------");
                printUpdate.println();
                printUpdate.println("Any order hasn't been placed yet.");
                printUpdate.println();
                printUpdate.println("-----------------------------------------------------------");
                printUpdate.println();
        }

        // if user tried to enter tracking number that is larger than the tracking number produced
        // it shows as error.
        else if(trackNum > trackingNumber) {
                ErrorMessageInvalidTNum(printUpdate);
        }

        else {
                boolean found = false;
                int index = 0;

                // we are going to traverse the orderHolder until we find the matching track number or,
                until
                // the track number at index exceed the target track number
                while(found == false && index < orderHolder.size()) {
                        int tempTNum = orderHolder.get(index).getTrackingNumber();
```

```
                    if(tempTNum == trackNum) {
                        found = true;
                    }
                    else {
                        index++;
                    }
                }

                // if loop terminates and the index was the size of the orderHolder which indicates that
                the order matches wasn't found
                // it is considered as a error and print out error message.
                if(index == orderHolder.size()) {
                    ErrorMessageInvalidTNum(printUpdate);
                }
                else {
                    String orderType;
                    String name = orderHolder.get(index).getFirstName() + " "+
                    orderHolder.get(index).getLastName();
                    String status = orderHolder.get(index).getStatus();
                    if(orderHolder.get(index) instanceof InStorePickUp) {
                    orderType = "In store pick up";
                    printUpdate.println("Order Type: "+orderType);
                    printUpdate.println("Name: "+ name);
                    printUpdate.println("Status: "+ status);
                }
                else if(orderHolder.get(index) instanceof ShippedOrder) {
                        if(orderHolder.get(index) instanceof WaitlistOrder) {
                            orderType = "Waitlist order";
                        }
                        else {
                            orderType ="Shipped order";
                        }
                        printUpdate.println("Order Type: "+orderType);
                        printUpdate.println("Name: "+ name);
                        printUpdate.println("Status: "+ status);
                }
            }
            printUpdate.println("-----------------------------------------------------------------");
            printUpdate.println();
            }
}
```

```
public static InStorePickUp createInStorePickUp(String[]customerInfo, String[]itemsList, PrintWriter printUpdate) {
        InStorePickUp order = new InStorePickUp();
        //InStorePickUp order;
        int tnum = trackingNumber;
        trackingNumber+=1;
        String fname = customerInfo[1];
        String lname = customerInfo[2];
        String location = customerInfo[3];

        //helper method1.
        boolean inStock = checkOrderedItems(itemsList);
        boolean Quantity;

        Random r = new Random();
        int n = r.nextInt(1);
        String status = "Processing Order";
        if (n == 0) {
                status = "Ready for Pickup";
        }
        if(inStock == true) {
                //helper method 2: check quantity
                Quantity = checkQuantity(itemsList);
                if(Quantity == true) {
                        ArrayList<Item> cart = new ArrayList<Item>();
                        for(int i = 0;i<itemsList.length;i++) {
                                Item searched = inventory.searchItem(itemsList[i]);
                                Item temp = inventory.putItemInOrder(searched);
                                cart.add(temp);
                        }
                        order = new InStorePickUp(cart, fname, lname, day, tnum, status, location);
```

```
            double totalCost = order.calculateCost();
            return order;
        }
        else {
            ArrayList<Item> place = new ArrayList<Item>();
            ArrayList<String>remove = new ArrayList<String>();
            for(int i = 0;i <itemsList.length;i++) {
                Item temp = inventory.searchItem(itemsList[i]);
                if(temp.getStockNo() == -1) {
                    remove.add(itemsList[i]);
                }
                else {
                    place.add(inventory.putItemInOrder(temp));
                }
            }
            // if arrayList of item is empty which means that there's no item that can be in
            the order,
            // the order is cancelled.
            if(place.isEmpty()) {
                printUpdate.println(fname +","+"Your order could not go through,
                please try again.");
                InStorePickUp bad = new InStorePickUp();
                bad.setTrackingNumber(-1);
                return bad;
            }
            else {
                printISPUpdate(fname,lname,remove,printUpdate);
            }
            order = new InStorePickUp(place, fname, lname, day, tnum, status, location);
            double totalCost = order.calculateCost();
            return order;
```

```
            }
}
else {
            ArrayList<Item> place = new ArrayList<Item>();
            ArrayList<String>remove = new ArrayList<String>();
            for(int i = 0;i <itemsList.length;i++) {
            Item temp = inventory.searchItem(itemsList[i]);
            if(temp.getStockNo() == -1) {
            remove.add(itemsList[i]);
}
else {
            place.add(inventory.putItemInOrder(temp));
                        }
            // if arrayList of item is empty which means that there's no item that can be in the order,
            // the order is cancelled.
            if(place.isEmpty()) {
            printUpdate.println(fname +","+"Your order could not go through, please try again.");
            InStorePickUp bad = new InStorePickUp();
            bad.setTrackingNumber(-1);
            return bad;
            }
                        else {
            printISPUpdate(fname,lname,remove,printUpdate);
            }
                        order = new InStorePickUp(place, fname, lname, day, tnum, status, location);
                        double totalCost = order.calculateCost();
                        return order;
            }
}
```

```
public static ArrayList<Order> createShippedOrder(String[] customerInfo,String[]itemsOrdered){
        ArrayList<Order> order = new ArrayList<Order>();
        ArrayList<Item> place;
        ArrayList<String>itemNeeded;
        ShippedOrder so;
        WaitlistOrder wo;
        //before we make order, we want to make sure if items in order are currently inStock or not
        //helper method 1.
        int tnum = trackingNumber;
        trackingNumber+=1;
        String fname = customerInfo[1];
        String lname = customerInfo[2];
        String shippingCompany = customerInfo[3];
        String waitlistPreference = customerInfo[customerInfo.length-1];
        boolean inStock = checkOrderedItems(itemsOrdered);
        // if it was true, next we need to make sure we have quantity of that item needed.
        // if boolean is false, then we will start creating order. whether it will be split or not is depending
        on the user
        boolean Quantity;
        if(inStock == true) {
                // helper method 2: check quantity
                Quantity = checkQuantity(itemsOrdered);
                //this should be possible only when items are in stock and have enough quantity.
                if(Quantity == true) {
                        // might be reusable as a method to truly just create shipped order instead of
                        decision making. line 120 to 132.
                        // we are not sure about placed date yet.
                        place = new ArrayList<Item>();
                        for(int i = 0;i<itemsOrdered.length;i++) {
                                //first we just search for the item if it's in the inventory,
                                Item searched = inventory.searchItem(itemsOrdered[i]);
                                //based on the search result, we call putItemInOrder, when this
                                method is involked, Item object in inventory will be removed
                                // and temp will save what we take out from inventory for the order.
                                Item temp = inventory.putItemInOrder(searched);
                                place.add(temp);
                        }
                        so = new ShippedOrder(place,fname,lname,day,tnum,"Processing
                        Order",shippingCompany);
                        so.calculateCost();
                        order.add(so);
```

```
            }
}
else {
            //this else block execute when we have item but we don't have enough of them.
            // and if user wanted to split the order, inside this if statement it create waitlist order
            and
            //shipped order
            if(waitlistPreference.equals("Split")) {
            // we save the tracking number temporary, and after saving the tracking number
            // we increment the tracking number by one for the next order to be placed.
            int tnum2 = trackingNumber;
            trackingNumber+=1;
            // these arraylist help us to separate items that are in the inventory that we can take out
            to put into
            // the shipped order and name of items that we need for the waitlist order to be
            completed.
            place = new ArrayList<Item>();
            itemNeeded = new ArrayList<String>();
            for(int i = 0;i<itemsOrdered.length;i++) {
                        Item temp = inventory.searchItem(itemsOrdered[i]);
                        if(temp.getStockNo() == -1) {
                                    itemNeeded.add(itemsOrdered[i]);
                        }
                        else {
                                    place.add(inventory.putItemInOrder(temp));
                        }
            }
            so = new ShippedOrder(place,fname,lname,day,tnum,"Processing
Order",shippingCompany);
            ArrayList<Item> split = new ArrayList<Item>();
            wo = new
            WaitlistOrder(split,fname,lname,day,tnum2,"Waitlisted",shippingCompany,itemNeeded)
            ;
            so.calculateCost();
            wo.calculateCost();
            order.add(so);
            order.add(wo);
}
else if(waitlistPreference.equals("Merge")) {
            //create only waitlist order
            place = new ArrayList<Item>();
            itemNeeded = new ArrayList<String>();
            for(int i = 0;i<itemsOrdered.length;i++) {
```

```
                        Item temp = inventory.searchItem(itemsOrdered[i]);
                        if(temp.getStockNo() == -1) {
                                itemNeeded.add(itemsOrdered[i]);
                        }
                        else {
                                place.add(inventory.putItemInOrder(temp));
                        }
                }
                wo = new
                WaitlistOrder(place,fname,lname,day,tnum,"Waitlisted",shippingCompany,itemNeeded);
                wo.calculateCost();
                order.add(wo);
        }
        return order;
}

//-----------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------

public static ArrayList<Order> createExpressOrder(String[] customerInfo,String[]itemsOrdered){

        ArrayList<Order> order = new ArrayList<Order>();

        ExpressOrder expro;

        WaitlistOrder wo;

        ArrayList<Item> place;

        ArrayList<String> itemNeeded;

        int tnum = trackingNumber;

        trackingNumber+=1;

        boolean inStock = checkOrderedItems(itemsOrdered);

        if(inStock == true) {

                boolean Quantity = checkQuantity(itemsOrdered);

                //this is in the situation we have enough stock to complete the order.

                if(Quantity == true) {

                        //create only express order

                        place = new ArrayList<Item>();

                        //double subtotal = 0;

                        for(int i = 0;i<itemsOrdered.length;i++) {
```

```
                    //we could definitely simplfy here.
                    //first we just search for the item if it's in the inventory,
                    Item searched = inventory.searchItem(itemsOrdered[i]);
                    //based on the search result, we call putItemInOrder, when this
                    method is involked, Item object in inventory will be removed
                    // and temp will save what we take out from inventory for the order.
                    Item temp = inventory.putItemInOrder(searched);
                    place.add(temp);
                }
                expro = new
                ExpressOrder(place,fname,lname,day,tnum,"Shipped",shippingCompany,notific
                ation);
                expro.calculateCost();
                order.add(expro);
            }
        }
        else {
            if(waitlistPreference.equals("Split")) {
                int tnum2 = trackingNumber;
                trackingNumber+=1;
                place = new ArrayList<Item>();
                itemNeeded = new ArrayList<String>();

                for(int i = 0;i<itemsOrdered.length;i++) {
                    Item temp = inventory.searchItem(itemsOrdered[i]);
                    if(temp.getStockNo() == -1) {
                        itemNeeded.add(itemsOrdered[i]);
                    }
                    else {
                        place.add(inventory.putItemInOrder(temp));
                    }
```

```
                }

                expro = new
                ExpressOrder(place,fname,lname,day,tnum,"Shipped",shippingCompany,notific
                ation);

                expro.calculateCost();

                ArrayList<Item>waitlist = new ArrayList<Item>();

                wo = new
                WaitlistOrder(waitlist,fname,lname,day,tnum2,"Waitlisted",shippingCompany,it
                emNeeded);

                wo.calculateCost();

                order.add(expro);

                order.add(wo);

        }

        else if(waitlistPreference.equals("Merge")) {

                place = new ArrayList<Item>();

                itemNeeded = new ArrayList<String>();

                for(int i = 0;i<itemsOrdered.length;i++) {

                        Item temp = inventory.searchItem(itemsOrdered[i]);

                        if(temp.getStockNo() == -1) {

                                itemNeeded.add(itemsOrdered[i]);

                        }

                        else {

                                place.add(inventory.putItemInOrder(temp));

                        }

                }

                wo = new
                WaitlistOrder(place,fname,lname,day,tnum,"Waitlisted",shippingCompany,item
                Needed);

                wo.calculateCost();

                order.add(wo);

        }

}

return order;
```

```
}


//*Helper Methods//
//-----------------------------------------------------------------------------------
//helper method 1:
public static boolean checkOrderedItems(String[]itemList) {
	boolean inStock = true;
	int i = 0;
	//this loop is going to run as long as each item are in the inventory, if it was not in the inventory
	then returns false
	while(inStock== true && i<itemList.length) {
		if(inventory.checkForItem(itemList[i])) {
			i++;
		}
		else {
			inStock= false;
		}
	}
	return inStock;
}

//-----------------------------------------------------------------------------------
//helper method 2: checking for quantity of item, checking to see if we have enough of that item.
public static boolean checkQuantity(String[]itemList) {
	boolean enough = true;
	int stockCount;
	int itemOrdered;

	// this loop does not stop in the middle because we need to make sure until the very endo of what
	is ordered.
	for(int i = 0; i<itemList.length;i++) {
		stockCount = inventory.checkCount(itemList[i]);
		itemOrdered = count(itemList,itemList[i]);
		// boolean remains true if stockCount was larger or equal to itemOrdered
		if(stockCount >= itemOrdered) {
			enough = true;
		}
		else {
			// if not then, it becomes false.
			enough = false;
		}
```

```
        }
        return enough;
        }


//-----------------------------------------------------------------------------------

//helper method 3: count how many of specific item we have in order

public static int count(String[]itemList,String item) {

        int count = 0;

        for(int i = 0;i<itemList.length;i++) {

                if(item.equals(itemList[i])) {

                        count++;

                }

        }

        return count;

}


//-----------------------------------------------------------------------------------

//helper method 4: print on out put file what is removed from order when InStorePickUp is created.

public static void printISPUpdate(String fname,String lname,ArrayList<String>remove, PrintWriter
printUpdate) {

        printUpdate.println("In store pick up order warning: ");

        printUpdate.println("Our sincere apology," + fname + " "+ lname);

        printUpdate.println("Items listed were not available and were removed from order: ");

        for(int i = 0;i <remove.size()-1;i++) {

                printUpdate.print(remove.get(i));

                printUpdate.print(",");

        }

        printUpdate.println(remove.get(remove.size()-1));

        printUpdate.println();

        printUpdate.println("-----------------------------------------------------------");

        printUpdate.println();
```

```
}
//-------------------------------------------------------------------------------------
//helper method 5: print daily order information to the company activity output file
public static void updateCompanyActivity(ArrayList<Order> orders, PrintWriter activity) {
        activity.println("Daily Activity Update Day " + day + ":");
        activity.println("-------------------------------------------------------------------");
for(int i = 0; i < orders.size(); i++) {
        activity.println(orders.get(i));
        activity.println("-------------------------------------------------------------------");
        }
}
//-------------------------------------------------------------------------------------
//helper method 6: print notifications for express orders to printUpdate if user wants notifications
public static void notify(ArrayList<Order> orders, PrintWriter printUpdate) {
        for (int i = 0; i < orders.size(); i++) {
                if (orders.get(i) instanceof ExpressOrder) {
                        Order temp = orders.get(i);
                        ExpressOrder temp2 = (ExpressOrder)temp;
                        boolean choice = temp2.getNotification();

                        if (choice == true) {
                                printUpdate.println("-------------------------------------------------------------
                        ");
                                String name = orders.get(i).getFirstName() + " " +
                        orders.get(i).getLastName();
                                int track = orders.get(i).getTrackingNumber();
                                String status = orders.get(i).getStatus();

                                printUpdate.println("Express Order notification for " + name + ":");
                                printUpdate.println("Status of Order #" + track + ": " + status);
                                printUpdate.println("-------------------------------------------------------------
                                ");
                        }
                }
        }
}


//-------------------------------------------------------------------------------------
```

```
//helper method 7: restock Inventory method
public static void restockInventory(String restockRequest) {
        // broke down the string into array of string
        String[]requestedItems = restockRequest.split("&");

        //search the item through the sample, and get the item information
        for(int index = 1;index < requestedItems.length;index++) {
                Item temp = inventory.searchItem(requestedItems[index]);
                Supplier s = temp.getSupplier();
                Item[] restock = s.sendItems(temp);

                for(int rs = 0;rs<restock.length;rs++) {
                        inventory.recieveItem(restock[rs]);
                }
                // get the supplier of the item and call send item, it will be saved into an array of items
                // transfer that item to inventory by using receiveItem method
}

//-------------------------------------------------------------------------------------
//helper method 8: this checks the inventory situation and update the waitlist so it can be completed as a
shipped order.
public static WaitlistOrder WaitlistOrderUpdate(WaitlistOrder check) {
        int size = check.getNotInStock().size();
        String[] temp = new String[size];

        //now we are allocating the name of items in ArrayList of waitlist order to array of String
        for(int t = 0;t < temp.length;t++) {
                temp[t] = check.getNotInStock().get(t);
        }

        //this condition checks if items are in the stock or not.
        if(checkOrderedItems(temp)) {
                //search for the item and get the item
                for(int t = 0;t<temp.length;t++) {
                        Item searched = inventory.searchItem(temp[t]);
                        Item place = inventory.putItemInOrder(searched);
                        //place the item into order
                        check.getOrderLine().add(place);
                        check.getNotInStock().remove(t);
                }
                check.setStatus("Shipped");
        }
```

```
		else {
			// if item is not in stock and the differnce of current day and orderplaced date was larger
			than or equal to 14 then it
			// returns empty waitlist order and in the main,
			if(day - check.getOrderPlacedDate() >= 14) {
				check.setStatus("Auto-Cancel");
			}
		}
		return check;
}

//-------------------------------------------------------------------------------------
//helper method 9: print auto cancel message: print out a cancel messege to let user know their waitlist
order had been cancelled.
public static void printCancelMessage(WaitlistOrder cancel, PrintWriter printUpdate) {
		printUpdate.println("-----------------------------------------------------------");
		printUpdate.println();
		printUpdate.println("Waitlist order status update: ");
		printUpdate.println(cancel.getFirstName() +" "+ cancel.getLastName() + ", your waitlist order
		had been cancelled.");
		printUpdate.println("-----------------------------------------------------------");
}

//-------------------------------------------------------------------------------------
//helper method 10: display the first message for user in output file for user
public static void printWelcomeMessage(PrintWriter printUpdate) {
		printUpdate.println("This is a notification window for customers: ");
		printUpdate.println("Thank you for choosing our service.");
		printUpdate.println("*****************************************************************");

}


//-------------------------------------------------------------------------------------
//helper method 11: warningOrderDelivered method
public static void printWarningOrderDelivered(Order order,PrintWriter activity) {
		String name = order.getFirstName() + " " + order.getLastName();
		int trackNum = order.getTrackingNumber();
		activity.println("\n**************************Warning**********************************");
		activity.println("\nThis order has been delivered and complete, therefore will be deleted from
		our system.");
		activity.println("Ordered by "+ name + ", tracking number: "+ trackNum);
		activity.println();
```

```
            activity.println("*************************************************************************");
        }


        //-------------------------------------------------------------------------------------

        //helper method 12: warningOrderPickUp method: this notifies the company that the in store pick up order
        // has been picked up.
        public static void printWarningOrderPickedUp(Order order,PrintWriter activity) {
            String name = order.getFirstName() + " " + order.getLastName();
            int trackNum = order.getTrackingNumber();
            activity.println("\n*************************** Warning *********************************");
            activity.println("\nThis order has been picked up and will be deleted from our system.");
            activity.println("Order by "+ name + ", tracking number: "+ trackNum);
            activity.println();
            activity.println("*************************************************************************");
        }

        //-------------------------------------------------------------------------------------

        //helper method 13: ErrorMessageInvalidTNum method: prompt user that the tracking number they entered is invalid.
        public static void ErrorMessageInvalidTNum(PrintWriter printUpdate) {
            printUpdate.println("************************ Warning ******************************");
            printUpdate.println("\nInvalid Tracking number, please check your tracking number and try again");
            printUpdate.println("\n************************ Warning ******************************");
        }

}
```

Input Files

Input File 1: Inventory.txt

Pandemic

4,Toilet Paper,1,4.00

2,Compact Desk,2,45.00

15,Toothbrush,3,1.00

3,Peanut Butter,4,3.50

7,Pen,5,0.50

4,Paper Towels,6,4.00

3,Napkins,7,3.00

2,Headphones,8,20.00

5,Wallet,9,90.00

2,Laptop,10,450.00

Input File2: UserInput.txt

MO

ISPU,John,Harris,Winter Park

Toilet Paper,Toilet Paper,Peanut Butter

MO

SO,Mary,Warren,FedEx,Merge

Toilet Paper,Compact Desk

MO

SO,Sal,Smith,FedEx,Split

Toilet Paper,Compact Desk

MO

ISPU,John,Harris,Winter Park

Toilet Paper,Toilet Paper

MO

EXPO,Sarah,Brown,UPS,true,Split

Toothbrush,

T

1

MO

EXPO,Jordan,Fox,FexEx,false,Merge

Compact Desk,Headphones

T

2

MO

ISPU,Larry,Hilfigger,Maitland

Laptop,Pen

MO

SO,Teddy,Rose,FedEx,Split

Toothbrush,Wallet,Paper Towel

MO

ISPU,Hiro,Sato,Japan

Toothbrush,Wallet,Paper Towel

MO

EXPO,Sarah,Brown,UPS,true,Split

Paper Towel,Wallet