CS 351: Process-Oriented Model Lab

Practice with Process-Oriented Modeling in DESMO-J

Overview

The goal of this lab is to gain experience implementing simulation models using a process-oriented perspective in DESMO-J. If you have not already done so, please download DESMO-J. You may also wish to read through the first several sections of the tutorial before getting started.

These exercises do not need to be turned in for credit, but you may find them helpful when working on the homework. Be sure to take advantage of this lab time to ask for help or clarification from the instructor and/or your peers.

To begin, create a new Eclipse project named `cs351-po-lab` (or anything else you prefer). Then drag and drop the DESMO-J `.jar` file into the `src` folder of your Eclipse project. When prompted, select "Copy files" (instead of linking). Once the `.jar` shows up in your project, right-click on it, select "Build Path", then "Add to Build Path". *Note:* You will need to use Java 8 (or higher) in your project in order to run your code.

The System

This exercise explores a small business-model, more specifically, the bar in La Crosse: Bodega Brew Pub. Bodega Brew Pub has two different parts to the building, the bar and the kitchen. Specifically for this lab, we are just going to focus on the kitchen part for simplicity sake. In order to recreate a model of Bodega, we need to understand what a typical day may look like. This model will also go into handling different states of the business depending on what day it is. In the beginning of the lab, we will cover a state as if the business does not have specials on different days, but may have different interarrival times. In a typical week in the kitchen, you can expect interarrival times to be the following:

- Sunday - 24 minutes
- Monday - 27 minutes
- Tuesday - 17 minutes
- Wednesday 25 minutes
- Thursday - 25 minutes
- Friday - 20 minutes
- Saturday - 20 minutes

The business opens up at 11 PM and closes at 1 AM, so when implementing this model, it is important to know you only need to simulate 14 hours per day, and since our simulation will cover a week, it will cover a total of 98 hours.

During a typical day at Bodega, you can expect customers coming in at their randomly distributed times (as explained above) and ordering random amounts of random foods. Some customers may choose an appetizer and an entree, while others may choose just an entree, and others may order two entrees, etc. Once the customer arrives, they walk to the kitchen and see if there is a cook available, if there is no cook available there would be a line until their order is taken, otherwise they would order. Usually at Bodega, there are two cooks on at any given time, sometimes one, but for our model we will assume there will be 2 cooks on at all times. Once a cook has taken their order, the cook will prepare their food and serve it to the customer, and so the cook will be unavailable to take another customer's order. Again, in the actual system at Bodega Brew Pub, a cook may take multiple orders, and then create a queue of orders that they cook, but for simplicity sake, our model will just serve one customer at a time. Each food has a cook time of their own, and if a

customer orders multiple items, the cook time will be the largest cook time that was found in all of their order.

It is important to note that the cook times listed below are uncertain times. Sometimes, raw ingredients to an item may run out and need to be prepared. Sometimes, depending on the cook's experience, the food may burn and need a longer time to cook. In our model, we will just cover the prep times without any uncertainty, but if you are interested in implementing this, then you can use a BoolDistBernoulli to generate a boolean on how the food may have burnt, or how an ingredient may need to be prepped.

Finally, once the cook has finished preparing their food, they serve the food to the customer and immediately move on to the next customer. These numbers listed below are from the system but are altered slightly for better readability. Expect all these numbers to be used in the lab.

Food	Price	Cost	Probability	Prep Time (in minutes)
Baked Artichoke Dip	$9.00	$3.50	0.10	5
BBQ Pork Flatbread	$9.00	$3.50	0.04	11
BBQ Pork Sandwich	$8.75	$3.25	0.08	4
BLT Sandwich	$6.00	$2.00	0.06	6
Build Own Salad	$9.00	$4.00	0.03	6.5
Chicken Bacon Flatbread	$10.00	$3.25	0.10	11
Chili Dog	$4.00	$1.00	0.03	6.5
Chips and Salsa	$5.50	$2.00	0.03	3
Crème Brulee	$5.00	$3.00	0.01	3
Cubano Sandwich	$10.5	$4.00	0.09	5
Grilled Cheese	$5.00	$2.00	0.06	6
Ozzie Sandwich	$10.50	$4.25	0.05	6
Pusan Sandwich	$11.25	$4.50	0.06	6
Reuben Sandwich	$10.50	$4.25	0.09	6
South By Southwest Sandwich	$8.00	$3.50	0.08	5.5
Tomato Basil Soup	$4.00	$1.50	0.05	1
Vegetarian Flatbread	$9.00	$3.50	0.06	11.5
Walnut Pesto Bruschetta	$7.5	$3.00	0.06	5

Exercise 1: Exploring a Process-Oriented Environment

The numbers listed above can be used to accurately predict one week of sales.
As like every simulation, there are questions we may want to ask about the system so that our model can find these values. Take the following questions,

- What is the average profit per hour?
- How much time does a cook (at any given time) spend their time preparing food for customers.

These questions specifically, we want to answer, but if there are any questions you can come up with and are curious about, try implementing a way to capture it. We have covered event-oriented perspectives to accurately simulate a system, so in this particular exercise, our goal is to model this system using a process-oriented perspective. A tutorial on using processes in DESMO-J can be found here Let us begin creating the model with our standard Model class, named `BodegaModel`, to represent our system and should extend DESMO-J's `Model` class. If you are using Eclipse, in the class creation dialogue box, set the superclass to `desmoj.core.simulator.Model`. This should automatically add several methods that need to be implemented, which we will fill in later.

Identifying Entities

Our model has been created and our next step in the process is to identify entities/attributes. This should be simple enough, although for those who have received service at Bodega Brew Pub should know that things differ slightly with the bar portion of the business. In this specific scenario, just focus on the kitchen in order to achieve what we want. Therefore our entities of interest are:

- Customer: An entity that is meant to receive service from a cook.
- Cook: An entity employed at Bodega Brew Pub (**initially started with pay of $10.50**) that takes the customer's order and cooks the order.

But similarly to an event-oriented perspective, we need to generate our customers, but instead of using an event to generate our customers, we will create an additional entity:

- CustomerGenerator: A make-believe entity that generates customers as the system runs.

Before creating Java classes to represent our entities, we'll first add a few `import` statements and global constants to the `BodegaModel` class, shown below:

```
import desmoj.core.simulator.*;
import desmoj.core.dist.*;
import desmoj.core.statistic.*;
import java.util.concurrent.TimeUnit;

/**
 * A process-oriented model of a small-business, Bodega Brew Pub.
 * @author <YOUR NAME HERE>
 * Last Modified: <TODAY'S DATE HERE>
 */
public class BodegaModel extends Model
{

    /** Model constants */
    protected static final int BUSINESS_HOURS = 14;
    protected static final double
            SUNDAY_INTERARRIVAL = 24,
        MONDAY_INTEARRIVAL = 27,
        TUESDAY_INTEARRIVAL = 17,
```

```
        WEDNESDAY_INTEARRIVAL = 25,
        THURSDAY_INTEARRIVAL = 25,
        FRIDAY_INTEARRIVAL = 20,
        SATURDAY_INTEARRIVAL = 20;
}
```

Being in a process-oriented model, we can still use DESMO-J's `Entity` class, but DESMO-J has a `SimProcess` class that we will be modeling our entities off of. This class provides use of some important methods, `activate`, `passivate`, and `hold`.

So next, create appropriately named classes for each entity which extend DESMO-J's `SimProcess` class. In Eclipse's class creation dialogue box, set the superclass to `desmoj.core.simulator.SimProcess`. You will need to add attributes and constructors for each class. By default, all DESMO-J model components require at least the following three parameters in their constructors:

- A reference to the `Model` that owns the component
- A name for the component (`String`)
- A `boolean` flag indicating if the component should be included in the simulation trace

(Some components also include an additional `boolean` flag for specifying if they should be included in the report.) This set of parameters is fine for the `Customer` constructor. Although, we will need to augment the parameters to fit another variable called 'foodSelectionDist' which accurately selects the food that the entity would like to order.

```
import desmoj.core.simulator.*;

public class Customer extends SimProcess
{

    /** Customer attributes */
    protected DiscreteDistEmpirical<Integer> foodSelectionDist;

    public Customer(Model owner, String name, boolean showInTrace,
   DiscreteDistEmpirical<Integer> foodSelectionDist)
    {
        super(owner, name, showInTrace);
        this.foodSelectionDist = foodSelectionDist;
    }
}
```

The variable, **foodSelectionDist** corresponds to the distribution that the customer should sample from (for a specific day). This does not need to be implemented exactly like this, and instead can be implemented in other ways. However, in this example, a foodSelectionDist is picked from a variable in the `BodegaModel`, called foodSelectionDistList. Initially, our `foodSelectionDists` in the `foodSelectionDistList` will all have the same exact values, which render all but one of them useless, but the reason we are implementing this now, is because Exercise 2 of this lab will be covering the idea that probabilities of foods may change from day to day.

Model Instance Variables: Queues and Distributions

All of our entities (SimProcesses) are finished, therefore all we have to do is setup the `BodegaModel` class before we build the entire process! The `BodegaModel` class has the standard methods that you may have seen in the Event-Oriented model. These methods being:

- `init` - Initializes all queues/distributions for the model.
- `doInitialSchedules` - Initially schedules all interarrivals.
- `description` - Returns a string of the description of the model.

Inside the `init` method, you will initialize all of your variables, queues, and distributions that are needed for the model to operate. Here is a list of the variables that you can choose to create:

- Counts:
  - Gross Profit
  - Items Sold
  - Cook Utilization
- Distributions:
  - Distributions dedicated for each day of the week. This is what was referenced above where an ArrayList holds a DiscreteDistEmpirical<Integer> for every day of the week. For now it is not necessary since all probabilities will be maintained, but in Exercise 2, there will be a portion where every day of the week, probabilities will be different and this is what can be helpful to track these probabilities.
  - Amount of food ordered by customer: There are multiple methods to implement this, I used a Bernoulli Distribution in a doWhile loop to implement this (in the Customer class). (`BoolDistBernoulli`) The distribution provides for the customer to randomly choose more than one item at a time. Some customers may order two items, some may order three, and some may order just one. Either way, we want to cover this situation by creating a do while loop, so the first loop is automatically counted for and one item is definitely ordered, then if the sample of this BoolDistBernoulli returns true, then the customer ordered a second item, and so on.
  - Interarrival distributions for each day of the week
  - Chance of balking distribution
- Queues:
  - Customers waiting.
  - Available cooks.

```
// counts to keep track of profit, items sold, and the amount of time
cooks are being utilized.
protected Count grossProfit, itemsSold, cookUtilization;

// queue to track which customer is in line to order food
protected Queue<Customer> kitchenOrderQueue;

// queues to track how often a cook is available to cook/take an order.
protected Queue<Cook> availableCooks, totalCooks;

// list of Empirical Distributions that hold the entire MENU's probability
on a given day of the week.
protected ArrayList<DiscreteDistEmpirical<Integer>> foodSelectionDistList
= new ArrayList<DiscreteDistEmpirical<Integer>>();

// sampling for chance of customer leaving the restaurant (due to length
of line).
protected DiscreteDistUniform balkChanceDist;

// sampling for how much food a customer may order.
protected BoolDistBernoulli foodAmountDist; // When this is initialized,
```

```
    it is initialized with probability 0.15.

    // sampling for interarrival times for every day.
    protected ContDistExponential sundayDist, mondayDist, tuesdayDist,
    wednesdayDist, thursdayDist, fridayDist, saturdayDist;
```

The initializations (in the `init` method) will be figured out by you, one exception being our foodSelectionDistList. Refer to these API's for assistance if needed: Queue, Count, DiscreteDistEmpirical, DiscreteDistUniform, BoolDistBernoulli, ContDistExponential.
Here is the code for initializing `foodSelectionDistList` assuming you are using the enum that will later be referenced. This can be altered if you choose not to use that enum.

```
String[] days = {"MO", "TU", "WE", "TH", "FR", "SA", "SU"};
for(String day : days)
    {
        switch(day)
        {
        case "MO":
            // EXERCISE 2
        case "TU":
            // EXERCISE 2
        case "WE":
            // EXERCISE 2
        case "TH":
            // EXERCISE 2
        case "FR":
            // EXERCISE 2
        case "SA":
            // EXERCISE 2
        case "SU":
            // EXERCISE 2
        default: break;
        }
        DiscreteDistEmpirical<Integer> foodSelectionDist =
            new DiscreteDistEmpirical<Integer>(this, "Discrete Empirical
    Distribution for the selection of food the customer wants on " + day, true
    , true);
        for (int i = 0; i < Food.MENU.length; ++i)
        {
            // Adds each item's probability off from menu with the ID that
    corresponds to i.
            foodSelectionDist.addEntry(Food.MENU[i].getId(), Food.MENU[i].
    getProb());
        }
        // Add foodSelectionDist to our list.
        foodSelectionDistList.add(foodSelectionDist);
    }
```

JAVA enums

In some programming languages, there is a type of class that is referred to as an enum. JAVA enums are provided for use of organization and ease of readability since they capture pre-defined constants. Refer to this documentation on enums for additional information
(**NOTE:** This entire section, as well as examples from this section, are NOT used in the actual model

we are creating. These are only examples that may be referred to, but are meant to communicate the importance of JAVA enums.)
Here is an example on how an enum could be added into a program.

```
protected enum DAY_OF_WEEK
{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

this can be used for easy switch cases or if/else statements. Take this implementation for example:

```
public DAY_OF_WEEK nextDayOfWeek(DAY_OF_WEEK day)
{
    switch(day)
    {
    case DAY_OF_WEEK.MONDAY:
        return DAY_OF_WEEK.TUESDAY;
    case DAY_OF_WEEK.TUESDAY:
        return DAY_OF_WEEK.WEDNESDAY;
    case DAY_OF_WEEK.WEDNESDAY:
        return DAY_OF_WEEK.THURSDAY;
    case DAY_OF_WEEK.THURSDAY;
        return DAY_OF_WEEK.FRIDAY;
    case DAY_OF_WEEK.FRIDAY;
        return DAY_OF_WEEK.SATURDAY;
    case DAY_OF_WEEK.SATURDAY;
        return DAY_OF_WEEK.SUNDAY;
    default:
        return DAY_OF_WEEK.MONDAY;
    }
}
```

This implementation exactly may not be the easiest way to handle a change of the day of the week, but provides organization and understanding without any need for explanation and commenting. Java enums, are essentially instantiated variables that provide ease of understanding, but they can also store information. The downside to enums is they act as static variables and therefore use more memory. Here is an example and implementation of storing data, where data is seen as a schedule for that day.

```
protected enum DAY_OF_WEEK
{
    MONDAY(8, 16),
    TUESDAY(8, 16),
    WEDNESDAY(8, 16),
    THURSDAY(8, 16),
    FRIDAY(8, 16),
    SATURDAY(0, 0),
    SUNDAY(0, 0);

    // enums can have their own unique instantiated variables
    private int start, end;

    // enums act exactly like a regular class, and may need a constructor.
    // They do not need a modifier, however (public, private, protected)
    DAY_OF_WEEK(int startOfDay, int endOfDay)
    {
        this.startOfDay = startOfDay;
        this.endOfDay = endOfDay;
    }

    // finally, enums can also have their own methods so they can be accessed.
    public String getStart() { return startOfDay; }
    public String getEnd() { return endOfDay; }
    public void setStart(int startOfDay) { this.startOfDay = startOfDay; }
    public void setEnd(int end) { this.endOfDay = endOfDay; }
}

// method that returns an ArrayList of days off in the week.
public ArrayList<DAY_OF_WEEK> getDaysOff()
{
    DAY_OF_WEEK[] daysOfWeek =
        {
         DAY_OF_WEEK.MONDAY,
         DAY_OF_WEEK.TUESDAY,
         DAY_OF_WEEK.WEDNESDAY,
         DAY_OF_WEEK.THURSDAY,
         DAY_OF_WEEK.FRIDAY,
         DAY_OF_WEEK.SATURDAY,
         DAY_OF_WEEK.SUNDAY
        };
    ArrayList<DAY_OF_WEEK> daysOff = new ArrayList<DAY_OF_WEEK>();
    for(int i = 0; i < daysOfWeek.length; ++i)
    {
        if(daysOfWeek[i].getStart() == 0 && daysOfWeek[i].getEnd() == 0)
            daysOff.add(daysOfWeek[i]);
    }
    return daysOff;
}
```

Food enum

Now that we have the basics of JAVA enums covered, we can apply them to our model. For this part, the table that was provided to you on page 1 is what will be used to create all of the instantiated variables. Additionally, we also want to keep track of how many of each item is sold, and therefore we should add another instance variable called `totalSold` and corresponding methods `sold()` and `getTotalSold()`. Here is a brief look at how our Food enum should look

```
public enum Food
{
    ART_DIP(9.00, 3.50, 0.08, 5),
    BBQ_FLAT(9.00, 3.50, 0.04, 11),
    // Remainder of the Menu and their respective values.

    // Specifically used to reference to the above instantiated enums in an
   easier manner when using distributions.
    public static final Food[] MENU =
        {
            ART_DIP,
            BBQ_FLAT,
            // Remainder of the instantiated enums from above.
        };
    private static int id = 0; // Used to create every enums FOOD_ID

    private final double basePrice, cost, baseProb, timeToCook; // basePrice
   is the original price
                                        // cost is the cost of the item to the
    business.
                                        // baseProb is the original
   probability
                                        // timeToCook is the time to cook the
   item.
    private double price, prob; // price is the current price (as if there may
    be a sale applied to it)
                     // prob is the current prob (as if a sale may affect how
   often it is purchased
    private int sales;       // total amount of sales
    private int FOOD_ID;     // Used to reference back to the MENU array. To
   set this, just use the static int from above 'id' and then set FOOD_ID to
   that, while simultaneously incrementing id.

    // CONSTRUCTOR
    Food(double price, double cost, double prob, double timeToCook)
    {
        this.basePrice = price;
        this.cost = cost;
        this.baseProb = prob;
        this.timeToCook = timeToCook;
        this.price = price;
        this.prob = prob;
        setId(); // This is the method that uses the static int 'id' to set
   FOOD_ID.
    }

    // Used to get a specific food item from the MENU when a customer orders.
```

```
    public static Food generateOrder(double sample)
    {
        return Food.MENU[(int) sample];
    }

    // Insert getters and setters here
    // For a specific setter: set the price and probability back to its
   default basePrice and baseProb, respectively.
}
```

Identifying Life Cycles

With all of our initializations and instance variables created, there is only one step left to work with (assuming our main method is set up) before our model is ready to simulate. This step is similar to creating event classes in an Event-Oriented model, but instead the entities 'interact' with each other to accomplish a life cycle.

Customers interact with each other by using three methods that come with DESMO-J's `SimProcess` class, these methods being: `activate`, `hold`, and `passivate`.

- `activate` - Wakes up a currently passivating entity.
- `hold` - Stops the process of an entity for timeToHold amount of time. args: timeToHold = double
- `passivate` - Stops the process of an entity for an arbitrary amount of time until activated again.

Let's start with the `Cook`'s `lifeCycle` method. With every cook, we intend to track their rate being subtracted from gross profits, and therefore that is all our `Cook` `lifeCycle` methods will consist of: updating our model's **Gross Profit** variable by \$11 per hour.

Next is the `CustomerGenerator`'s `lifeCycle` method. This class is made up to be the backbone of the simulation. Without a generator for the dynamic entities in the model, there would be no simulation! Think of any entity that comes and goes to and from the model as any entity that needs a generator. Since our model only has one dynamic entity, a customer, we create an entity called `CustomerGenerator` where its `lifeCycle` will call various helper methods to accomplish this task. Look back to our enums example. We used an implementation called `changeDay` which we will be using now, although we do not have to implement it as an enum. This can easily be done by creating a group of integer variables named after each day of the week initialized from 1 to 7. Then the method `changeDay` can take one integer as an argument (the current day), and use a switching statement to return the next day.

The best way to implement the CustomerGenerator's `lifeCycle` method is to think of it as a normal day at Bodega Brew Pub and the behavior of the system. The model we are starting on a Tuesday, and so we need to create an integer variable called **currentDay** and initialize it to **TUESDAY**. It is optional, however helpful, to send a trace note to the model just for debugging purposes so we know what day our model is on when the customer is generated.

Since our Customer Generator will be generating customers constantly until the simulation is over, we can do a simple **while(true)** loop. In this loop, it is important to get our interarrival time before anything else, and since we have multiple days in our model that behave differently, we should switch between the day's sample times. So, for organization, write a method called `getInterarrival` that takes two arguments, a BodegaModel and an integer that corresponds to the day, and returns one double that is the interarrival time.

```
public double getInterarrival(BodegaModel model, int day)
{
    final int MONDAY = 1, TUESDAY = 2, WEDNESDAY = 3, THURSDAY = 4, FRIDAY =
   5, SATURDAY = 6, SUNDAY = 7;
    switch(day)
    {
        case SUNDAY:
            return model.sundayDist.sample();
        // ... every other day of the week.
    }
}
```

Now that we have our interarrival time, all we have to do is `hold` the Customer Generator for that length of time, and once that time has passed, the generator should create and `activate` a new customer only to immediately check if the day is over. Refer back to to the code above where there was a final static integer called BUSINESS_HOURS. Use the expression (60 * BodegaModel.BUSINESS_HOURS) * day) to compare to the model's present time to check if the day is over with.

IMPORTANT: Since our model's day ranges between 1-7, our model can only simulate one week at a time, and if anything more is wanted to simulate, then the above expression must be changed or a Repetition model must be created.

Finally, the longest `lifeCycle` method will belong to the Customer. First, we will reference our enums example one more time to create an enum, which will consist of our menu. Call this enum `Food`. On the first page of the lab, there was a table that consisted of all of the attributes that are needed for tracking the menu items at Bodega Brew Pub. Since our customers need to randomly choose what food they want, we need to track the foods bought and have it easily accessible to the customers. You can implement this however you want, but the enum examples from above would be the most plausible way to complete this part. A skeleton of what you can use is provided above in the Food enum section.

Since most of the simulation will be within the Customer `lifeCycle` method, it's important to know what exactly we expect our Customer to do in our model, and who they may interact with. Refer back to the first page where Bodega Brew Pub's day-to-day schedule consisted of. Here is some pseudo-code to get you started.

```
// LIFE CYCLE METHOD:
    // Let X be a boolean that represents the decision of the customer balking

    /*
    Set X to the boolean variablee: sample of the Discrete Distributed
   Uniform initialized in BodegaModel (as an integer) <= size of the order
   queue.
    */

    /*
    [HINT: Distributions return a Long (the wrapper class) which has a method
   that returns an integer value.]
    */

    // If X is true, the customer has left

    // If X is false, continue service

    /*
    ... Service consists of checking if the customer can be waited on now.
    */

    /*
    ... Once the customer is waited on, generate their order. (Remember that
   customers have a distribution to check if they ordered more than one food
   item.)
    */

    /*
    ... Hold the customer for however long their longest Food item takes to
```

```
    cook and set the cook to busy.
     */

     /*
     ... Once activated again, update model variables and end service with this
      customer while simultaneously notifying the next customer that they can
    proceed with service.
     */
```

Main method

There is nothing too much to say for the main method, and so it will be provided to you with comments explaining what each line(s) of code does.

```
public static void main(String[] args)
{
    Experiment.setReferenceUnit(TimeUnit.MINUTES);
    // Creates our BodegaModel model.
    BodegaModel model = new BodegaModel(null, "Bodega Brew Pub Model", true,
   true);
    // Creates an experiment to attach to the model.
    Experiment exp = new Experiment("Bodega Brew Pub Model");
    // Setting a seed is good for debugging. 89 is just an arbitrarily chosen
   number.
    exp.setSeedGenerator(89);
    // Connect model to experiment
    model.connectToExperiment(exp);
    exp.setShowProgressBar(false);
    // Reference variables for readability
    int minutesInBusinessDay = BUSINESS_HOURS * 60;
    int daysToSimulate = 7;
    // Stop the simulation after so much time (98 hours for one week)
    exp.stop(new TimeInstant((minutesInBusinessDay * daysToSimulate), TimeUnit
   .MINUTES));
    exp.tracePeriod(new TimeInstant(0, TimeUnit.MINUTES),
        new TimeInstant(minutesInBusinessDay * daysToSimulate, TimeUnit.
   MINUTES));
    exp.debugPeriod(new TimeInstant(0, TimeUnit.MINUTES),
        new TimeInstant(minutesInBusinessDay * daysToSimulate, TimeUnit.
   MINUTES));
    // Begin simulation
    exp.start();
    exp.report();
    exp.finish();
    // OPTIONAL: Print all of the units that were sold.
    DecimalFormat df = new DecimalFormat("#0.00");
    double totalProfit = 0;
    int totalSold = 0;
    for (int i = 0; i < Food.MENU.length; ++i) {
        System.out.println(Food.MENU[i] + " - Amount sold: " + Food.MENU[i].
   getSales() + ", Total profit: $"
            + df.format(Food.MENU[i].totalProfitFrom()));
        totalProfit += Food.MENU[i].totalProfitFrom();
        totalSold += Food.MENU[i].getSales();
    }
    System.out.println("Total Units Sold: " + totalSold + ", Total Profit: $"
   + df.format(totalProfit));
}
```

Short Exercise: Providing more ambiguity

As mentioned severals time in Exercise 1, systems can be very difficult to simulate because of how much information they need. Although, Exercise 1 is all you need to understand how a process-oriented model works, this exercise can show you how difficult a simulation can get with the more chaos and randomness that ensues.

Bodega Brew Pub is a kitchen that can create a lot of this random activity with the bar as well, for example, some customers may want a heavy food with a heavy beer that is temporarily on top. For this exercise, we won't be covering these chaotic activites, but instead the randomness of how probabilities of foods are affected when there are sales and specials of food.

Recall that we had an ArrayList that held multiple distributions for foods. As of right now, our ArrayList (assuming that is what you use in this lab) is currently holding repititive values . This can be altered by adding code to our switch statement. First, I will give you a new table corresponding to our prices/probabilities.

Food	Price	Cost	Probability	Prep Time (in minutes)
Baked Artichoke Dip	$9.00	$3.50	0.10	5
BBQ Pork Flatbread	$9.00	$3.50	0.04	11
BBQ Pork Sandwich	$8.75	$3.25	0.08	4
BLT Sandwich	$6.00	$2.00	0.06	6
Build Own Salad	$9.00	$4.00	0.03	6.5
Chicken Bacon Flatbread	$10.00	$3.25	0.10	11
Chili Dog	$4.00	$1.00	0.03	6.5
Chips and Salsa	$5.50	$2.00	0.03	3
Crème Brulee	$5.00	$3.00	0.01	3
Cubano Sandwich	$10.5	$4.00	0.09	5
Grilled Cheese	$5.00	$2.00	0.06	6
Ozzie Sandwich	$10.50	$4.25	0.05	6
Pusan Sandwich	$11.25	$4.50	0.06	6
Reuben Sandwich	$10.50	$4.25	0.09	6
South By Southwest Sandwich	$8.00	$3.50	0.08	5.5
Tomato Basil Soup	$4.00	$1.50	0.05	1
Vegetarian Flatbread	$9.00	$3.50	0.06	11.5
Walnut Pesto Bruschetta	$7.5	$3.00	0.06	5
Carnitas Tacos*	$4.00	$2.00	0.00	8.5
Fish Tacos*	$4.00	$1.50	0.00	8

Here is a list of each day and their respect sales/specials.

- Monday - BBQ Pork Sandwich for 7.00: Probability increased by 0.08
- Tuesday - Buy one get one free (ALL ITEMS): Increase probability of foodAmount by 0.40 and allow every even item bought to be free.
- Wednesday - Corned Beef sandwiches -$2: Increase probability of Ozzie, Reuben, Pusan by 0.02, 0.04, 0.02 respectively.
- Thursday - Carnitas Tacos: Allow Carnitas Tacos to be sold for their base prices at probability 0.08.
- Friday - Fish Tacos: Allow Fish Tacos to be sold for their base prices at probability 0.08.
- Saturday - Chili Dogs for $3: Increase probability of Chili dog by 0.04
- Sunday - No specials or sales: defaults

First, we need to edit our Customer `lifeCycle` method so Tuesday specials are covered. We can start this by making a second BoolDistBernoulli in our model class called `foodAmountDistTues` and initializing it in the init method with 0.55 instead of 0.15. In the `lifeCycle` method, we need to constantly check if the customer is ordering on a Tuesday. You can simply do this by declaring and initializing a boolean called `isTuesday` to `foodSelectionDist == model.foodSelectionDistList.get(1)` The reason we get the second element from our `foodSelectionDistList` is because we initialized the second element to Tuesday's foodSelectionDist. We can then use this `isTuesday` variable to set a local variable called `amountFoodDist` to foodAmountDistTues or the regular foodAmountDist that we initialized in Exercise 1. Inside of our loop (presumably a doWhile statement), we can check if the item we generated is an even item. If it is, then we should not call the `sold` from that food item, because then the profits are taken into consideration. Instead, we should make a new method in our Food enum, called `free` which increases the amount of units sold, but only affects the cost value of our system, and not the profit.
After that, all that needs to be done is edit the switch statement in our `BodegaModel` class. Set each of the cases in the switch statement to their respective sales. Here is the first part of the code to get you started.

```
switch(day)
{
Food.MENU[Food.CARNITAS.getId()].setProb(0); // Carnitas are only served on
    Thursdays.
Food.MENU[Food.FISH_TACOS.getId()].setProb(0); // Fish tacos are only served
    on Fridays.
case "MO":
  Food.MENU[Food.BBQ_PORK.getId()].setProb(Food.BBQ_PORK.getProb() + 0.08); //
    BBQ Pork is on special.
  Food.MENU[Food.BBQ_PORK.getId()].setPrice(7);
  break;
case "TU":
  Food.MENU[Food.BBQ_PORK.getId()].defaultBases() // Set BBQ Pork sandwich
    back to daily price/prob.
  break;
case "WE":
  // WEDNESDAY SPECIAL
case "TH":
  // THURSDAY SPECIAL
case "FR":
  // FRIDAY SPECIAL
case "SA":
  // SATURDAY SPECIAL
case "SU":
  // SUNDAY SPECIAL (technically no special)
default: break;
}
```

Once that part is done, your code should be ready to handle specials and sales for the week.