Burrito Brothers Software

Design Documentation

CISC 640

Fall 2016

Professor: Dr. Gregory Simco

Submitted by: Anabetsy Rivero

Nova Southeastern University

Table of Contents

1. Introduction

The present document specifies the requirements, design, and implementation of Burrito Brothers. Burrito Brothers is an application written in Java 8 to satisfy the requirements specified in the Operating Systems CISC 640 course of Nova Southeastern University's Master's of Science in Computer Science. This application is an expanded version of the well known sleeping barber problem first proposed by Dijkstra in 1965. (Dijkstra, 1968) Burrito Brothers is designed to simulate customers entering, waiting, getting burritos, paying, and leaving Burrito Brothers, a shop located in Gainesville, Florida. Because the customers act like producers and the servers act like consumers, this exercise can be solved using semaphores for concurrency control.

1.1 Problem Requirements

- Burrito shop has a maximum capacity of 15, so only 15 customers can sit in the waiting area at any given time
- A customer can have an order of 1 to 20 burritos
- Each customer is an independent thread
- If the shop is full (i.e there are 15 customers waiting inside), the next customer leaves without entering the shop
- The shop has a waiting area where customers sit and wait until their turn to be served.
- The waiting area is organized in order of shortest order next
- There are three burrito servers with infinite supplies each
- Each server is an independent thread

- There are three shared counter locations
- Each server can only make three burritos at a time, therefore, customers who have more than three burritos, must go back in the waiting area and so on until their entire order is completed
- A server can only serve one customer at any given time
- There is one shared register so only one customer can pay at any given time
- After paying, the customer leaves the shop

1.2 Approach

This solution to burrito brothers consists of a single Java application where each server and each customer is a separate thread. Java's semaphore class is used to control access to the critical areas. In this application, customers arrive at the shop and try to enter if there is room in the waiting area. The waiting area, the three counters, the servers, and the cash register are represented by semaphores. Upon entering, the customer sits at the waiting area and then he is placed in a linked list (***customersWaiting***), which is controlled by a binary semaphore named ***customers***. The specific design decisions about the semaphores appear in the semaphores section of the documentation. The waiting area is organized by smallest order next so every time a new customer enters the waiting area, the linked list that represents the waiting area is sorted.

Customers are created by obtaining input from a text file, where the customer id (***custId***) appears in the first position and the order size (***numBurritos***) appears on the second position. The customer id and the number of burritos are separated by an empty space. When the program reads each line, it places the customer id at index 0 of a string array named ***line.*** Each customer thread is created from this array and each thread is then started.

The servers are represented by threads that are created from a string array which contains their names. By looping over the array, the server threads are created one at a time and subsequently started.

2. Data and Algorithm Design

2.1 Design of key data structures

This section of the document describes the design decisions for the use of semaphores to control concurrency, as well as threads, and a linked list data structure to hold waiting customers. In addition, it expands on the methods from the Semaphore class, the Thread class, and the Linked List class as well as other methods created for this project.

2.1.1 Semaphores

2.1.1.1 Overview of binary and counting semaphores used

The Java 8 Semaphore class was used to represent the waiting area, counters, customers, and cash register. Because the maximum shop capacity is fifteen, the waiting area was designed as a counting semaphore with fifteen available permits. When a customer tries to acquire the waiting area, the program checks if there are available permits, and if there are any, it allows the customer to enter the shop.

In addition to counting semaphores, binary semaphores can be used in a similar fashion as mutex locks. (Downey, 2005)For this reason, a binary semaphore was used to control access to the linked list containing the waiting customers. Once a customer has entered the waiting area, he acquires the customers semaphore, indicating that a customer is present and must be added to the linked list while excluding other threads from accessing the list at the same time. Once the customer has been succesfully added to the list, he releases the customers semaphore to signal the server that a customer is ready to be served. The next available server then acquires the customers semaphore, thus locking access to the linked list, takes the first customer on the list and makes three burritos at a time.

The counters were designed as a counting semaphore with three available permits because there are three counter locations where the servers can make burritos for the customers. Once a customer is finished waiting, he acquires a counter if one is available and releases his hold on the waiting area, thus increasing the number of available permits in the waiting area and decreasing the number of available permits on the counters. After the customer has succesfully acquired a counter, he is ready to be served and the signal to the server allows the next server to take the next available customer in the linked list.

The cash register was designed as a binary semaphore initialized with one permit so a customer who needs to pay can acquire this permit, pay for his burritos, and leave the shop. The cash register semaphore was included as part of the pay_burritos() method, which will be described in the methods sub-section of this section.

2.1.1.2 Methods of the Java Semaphore class used

acquire()

release()

From the Jave Semaphore class, two methods were used to implement Burrito Brothers. The methods used were ***acquire*** and ***release***, which are derived from the original methods ***P*** and ***V*** proposed by Dijkstra. (Downey, 2005)When a thread, or a customer in the case of burrito brothers, acquires a semaphore, be it a binary or a counting semaphore, the semaphore's available permits decrease by one. When a thread releases a semaphore, the number of available permits increases and another thread may acquire that permit.

2.1.2 Threads

The Thread class from Java 8 was used to simulate the customers and servers in the application. The classes ***Customer*** and ***BurritoServer***, which contain the methods for the customers and servers, were implemented as instances of the Thread class and both implement the Runnable interface. Java's Runnable interface is an interface that must be implemented by any class whose instances will run as independent threads. ("Runnable (Java Platform SE 8). (n.d.).
,")In Burrito Brothers, each line from the text file "customers.txt" was placed in a string array (***line***) containing the customer id (***custId***) and the number of burritos in the order (***numBurritos***). Each customer thread was then created from this array as an instance of the Customer class, which implements the Runnable interface. When the start method of the customer thread is called in main, the run method is called for each thread and each customer tries to enter the burrito shop.

Similarly, the servers are created as independent threads which are instances of the BurritoServer class, which also implements the Runnable interface. The server threads instead are constructed from a string array that contains their names. By giving each server thread a different name, we can trace which server is serving which customer at any given time. Each server thread is either serving a customer (removing a customer object from the linked list) or waiting for a customer.

2.1.2.1 Methods used

start()

sleep()

run()

toString()

get_burritos()

pay_burritos()

leave_shop()

make_burritos()

In this implementation of Burrito Brothers, the ***start()*** and ***sleep()*** methods of the Java Threads class were used. The ***start()*** method was used to start each thread (both customers and servers), which calls the thread's ***run***() method. In this application, each thread has a custom designed ***run()*** method, whose details appear in the pseudocode section. In the ***run()*** method of the Customer class, the customer tries to acquire the waiting area if a spot is available, then he is placed in the linked list ***customersWaiting*** and after the release() method is called on the customers semaphore, the customer gets his order (or at least three burritos from his order total) fulfilled by the next available server. While the customer is being served (approximately), the method ***get_burritos()*** is called. This method consists of a call to the ***sleep()*** method, where the customer thread sleeps for 1000 milliseconds. In a real life implementation, the customer thread would not be sleeping but this time elapsed, for the purposes of Burrito Brothers, simulates the customer waiting at the counter while the server makes his burritos. Once the order (or three of his burritos) are made, the ***pay_burritos()*** method is called. In this method, the customer acquires the cash register (***register***) semaphore, sleeps for another 2000 milliseconds, and then releases the register. After this step, the ***leave_shop()*** method is called and a message is printed.

Setter and getter methods were used for the customer id and the number of burritos. The ***toString()*** method was overriden in order to return the customer id and the number of burritos in string format. All cases, including customers who have orders of more than three burritos and who need to come back to have their full order fulfilled, make use of the same methods. Descriptions of those use cases will be discussed elsewhere.

In addition, the BurritoServer class also had an overriden ***toString()*** method to return the name of each server. The ***sleep()*** method was also used in a similar manner to the usage in the Customer class. It is important to note that the ***sleep()*** method was used while each server was making burritos as part of the ***make_burritos()*** method but in real life this would not be the case. It would have been nice to see the server adding steak, lettuce, tomatoes, etc but the output would have been much longer so this part was left to the user's imagination.

2.1.3 Linked List

In Burrito Brothers, the Java Linked List class was used to implement a structure to hold the waiting customers. Because the customers act as producers and the servers act as consumers, it is important that the linked list (***customersWaiting***) be protected by some thread safe mechanism. This could also be accomplished with an Array Blocking Queue, which is a thread safe data structure of the Java language. While one of my implementation efforts included such a structure, it proved difficult to use in this project because the queue could not be sorted in a shortest-order-next fashion. Otherwise, this could be a useful tool for high level implementation of concurrent solutions that do not involve semaphores. However, the requirements of this project specified that semaphores be used to control concurrency. Because the linked list is a dynamic data structure, it can be useful for this project.("Class LinkedList<E>,")

2.1.3.1 Methods used

add()

pollFirst()

Collections.sort(customersWaiting, new Comparator<Customer>()

The ***add()*** method was used to add newly arrived customers to the linked list of waiting customers. In contrast, ***pollFirst()*** was used to retrieve and remove the first object stored in the linked list (i.e. the customer with the smallest number of burritos). The sort methdo was used to sort the linked list in order of smallest number of burritos first.

2.2 Design of algorithms

This section describes the design of algorithms used in this solution of Burrito Brothers. Three classes were created to implement Burrito Brothers: Main, Customer, and BurritoServer. The following sub-sections further explain the features and design decisions of each class.

2.2.1 Main Class

The Main class begins with the “Welcome to Burrito Brothers” message. It takes the input from the text file “customers.txt” where the information for each incoming customer appears and places each customer in a string array. This string array is important because it facilitates a more automatic way of adding as many customers as appear in the text file and passing their customer ids (***custId)*** and order sizes (***numBurritos***) as parameters to the new Customer object (***customer***)being created. By using a text file as input, a user can simply change the sizes of the orders and/or add or delete customers, which makes it convenient to test. In this fashion, the first instance of the Customer class is created in Main from the input in the text file. If the number of burritos of the incoming customer is greater than zero and the waiting area still has space for at least one customer, the new customer is added to the linked list (***customersWaiting***), which is also created in Main. Subsequently, as the customers are added, the list is sorted in order of smallest order next. Even though sorting the list repetedly is a costly operation, accessing the linked list and removing the first object in it has a complexity of O(1), which is appealing given the number of times the servers need to remove an object from the list. After sorting, the customer threads are started.

Additionally, in Main there is another string array that contains the names of the servers. By looping through this array, the Server object (***server***) is created and then the server threads are started.

2.2.2 Customer Class

The Customer class is used to handle the methods related to the Customer objects created from the text file. This class implements the Runnable interface, which allows the objects of the

Customer class to run as independent Java threads. A boolean variable ***notServed*** is defined in the scope of the Customer class to keep track of the customers' status. In addition, an integer variable ***freeSeats*** is also defined to keep track of the number of free seats remaining in the waiting area. Each Customer has a customer id (***custId***), an order of n number of burritos (***numBurritos***), and a linked list (***customersWaiting)*** as parameters. The methods used in this class and others are discussed in the methods sections. The algorithm for the ***run()*** method is as follows:

```
public void run(){
while(notServed){
        if (there is room in waiting area){
                acquire waiting area
                decrease the number of free seats
                print message about customer x who has entered waiting area with order of n
                burritos
                release waiting area //get up to go up to counter
                acquire counters
                acquire customers //this acts as a mutex lock to add customer to list
                add customer to linked list
                sort list in order of smallest number of burritos next
                release customers //let servers know there are customers waiting to get burritos…
                get_burrito //method to simulate waiting at the counter while burritos are made…
                release counters //leave counter after order (or 3 burritos) are made
                pay_burritos //acquire cash register, sleep(2000), release cash register
                leave_shop//print message about customer leaving the shop…



        if( order is greater than 3 burritos){
                decrease numBurritos by 3
                acquire waiting area if a spot is available
                decrease the number of free seats
                acquire customers
                add customer to linked list
                release waiting area
                release customers //let servers know there are more customers waiting…
                }
        else if( order is between 1 and 3){
                decrease numBurritos by 3
                        if( order is less than zero burritos){
                                notServed= false;
                        }
                        else {
                                acquire waiting area
                                decrease the number of free seats
                                acquire customers
                                add customer to linked list
                                release waiting area
                                release customers// let servers know there are more customers
                                waiting…
                                acquire counters
                                get_burrito //method to simulate waiting at the counter while
                                burritos are made…
                                release counters
                                pay_burritos //acquire cash register, sleep(2000), release cash
                                register
                                leave_shop //print message about customer leaving the shop…
                        }
                }
        else {
                print message if shop is full //let customer x that there is no room and he will have
                to come back later
                notServed = true //this allows the customer to come back to main loop and keep
                trying to enter the shop
                 }
            }
        }
```

<u>2.2.3 BurritoServer Class</u>

The BurritoServer class is a class that also implements the Runnable interface for the same reasons that the Customer class implements it. As part of this class, each server thread has a name and access to the linked list (***customersWaiting***) by means of the customers semaphore. The ***run()*** method allows the next available server thread to access the next waiting customer in the linked list. The algorithm for this method appears below.

```
public void run(){
        while(there are customers waiting){
                acquire customers
                make_burrito//print name of server making burritos for customer x . Server thread
                sleeps for 500 ms
                release customers
        }
}
```

3. Object Oriented Design

3.1 UML Diagrams

1.2.1 State Diagrams: Customer and Server

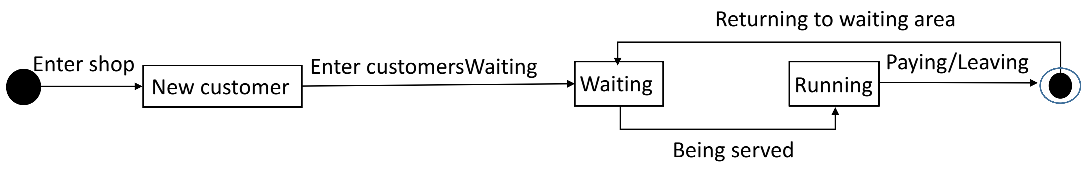


Fig 1: State diagram depicting the lifetime states of each customer thread in Burrito Brothers. At the beginning, a new customer thread is created for each line in the text file. Each customer can enter the waiting area and wait for the next available server. The customer thread runs while burritos are being made. The customer can pay and leave the shop after burritos have been made. However, if the order is not completely fulfilled, the customer goes back to waiting for another server and so on.

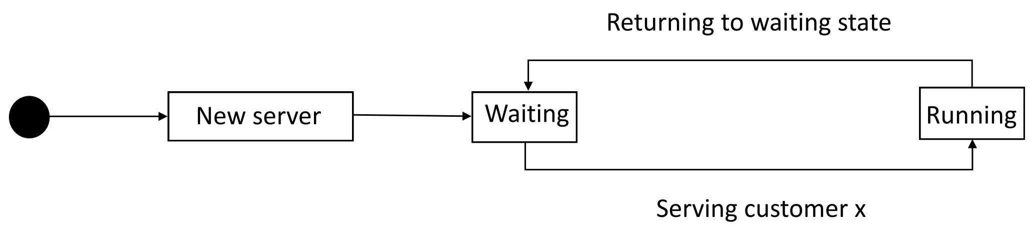


Fig 2: State diagram depicting the lifetime of each server thread in Burrito Brothers. At the beginning a new server thread is created for each of the names is the SERVERS array. Each server threadis either waiting for customers or running (serving a customer). The cycle continues while there are customers to be served.

1.2.2 Use Case Diagram

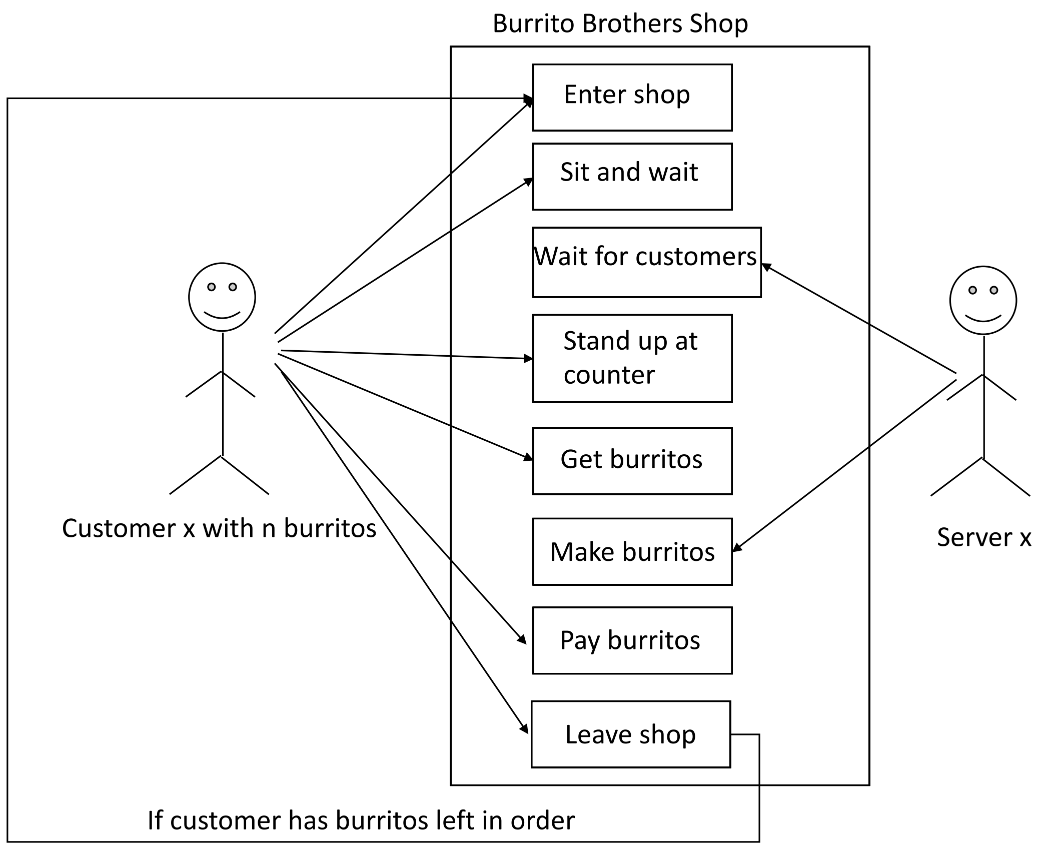


Fig 3: Use case diagram depicting the different use cases for the customers and servers in Burrito Brothers. A server is either waiting or serving a customer. A customer can enter the shop, sit in the waiting area, stand up at counter, get burritos made, pay for burritos, and leave the shop. If the entire order is not fulfilled at once, that is, if the customer origiannly had an order of more than three burritos, the customer can re-enter the shop until the order is completely fulfilled.

3.1.3 Class Diagram

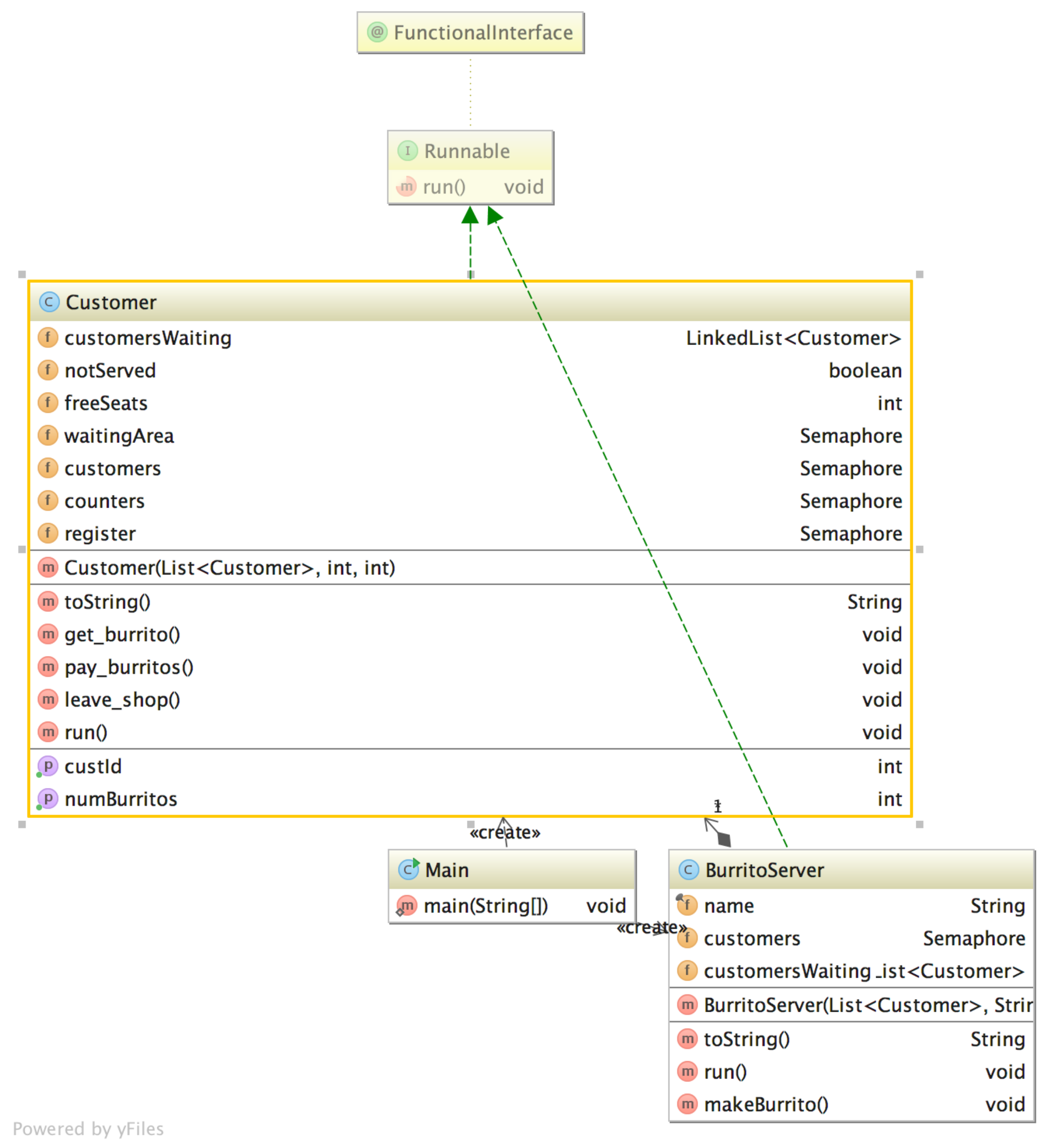
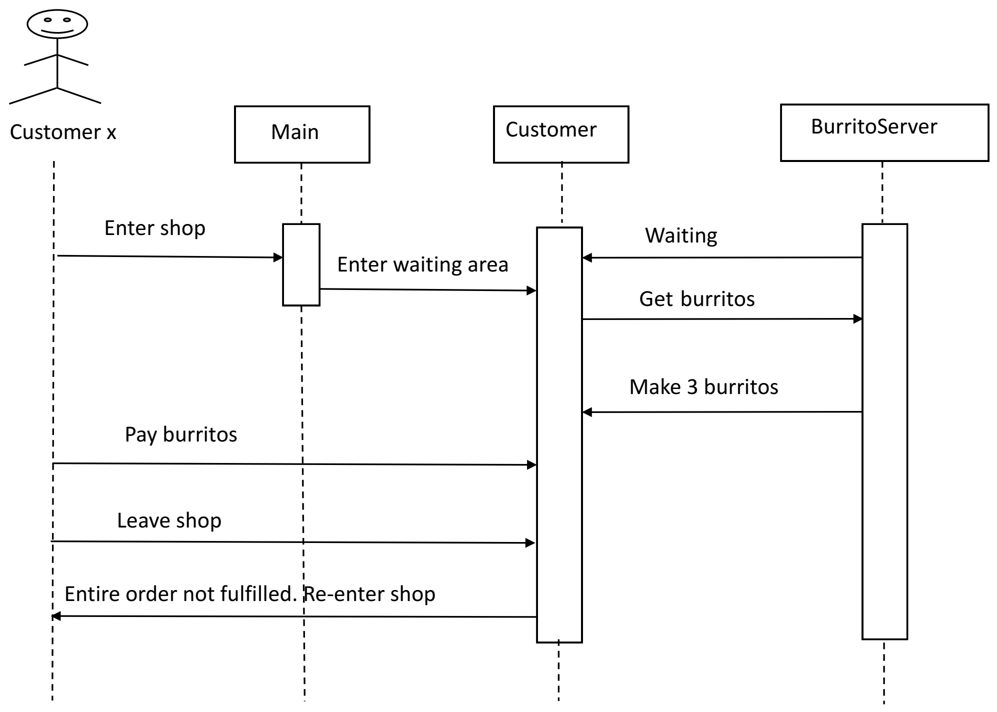


Fig 4: Class diagram depicting the relationships between the different classes in Burrito Brothers. The green dotted arrow means that a class implements an interface (i.e. BurritoServer implements the Runnable interface). Aso shown are the different methods and variables used in each class. This diagrams was produced usign IntelliJ IDEA Ultimate Edition 2016.

1.2.3 <u>Sequence Diagram</u>



1.3 <u>Testing</u>

In order to test Burrito Brothers there are several things we must do. First, we must test that every line in text file “customers.txt” results in a new customer with the corresponding customer id (***custId***) and order size (***numBurritos***). For instance, consider the following text file input:

```
1 5
2 3
3 1
4 6
5 2
6 4
7 1
8 3
9 5
10 1
11 2
12 3
13 4
14 1
15 2
```

- We know that the program is supposed to create a new customer from every line, where the first integer is the ***custId*** and the second integer is the ***numBurritos***. This can be tested by simply writing output stating the custId and the order size of each customer created. In this case, we must also make sure that the numbers returned are integers. Subsequently, we must test that the customer threads are started correctly. In the current implementation, these tests were conducted.

- Likewise, we must check that the servers are correctly created from the SERVERS string array containing their names. We should also check that the server threads are started correctly. This is confirmed by printing the name of the server currently serving a customer (BurritoServer class). Cases where something was incorrect led to printing of a single server name serving all customers in the shop, or printing no server names at all.
- Furthermore, we must test and understand how the semaphore methods work and their correct use in this program. I started by testing the acquire and release methods by printing the number of available permits for each semaphore used. This allowed me to understand and prove to myself that the acquire method call causes a decrease on the number of available permits, while the release method call causes an increase in this number.
- In order to test whether each customer has been correctly added to the linked list (***customersWaiting***), we must print the contents of the list before and after adding customer objects into it. If correctly implemented, this list will appear empty before any customers are added and the size of the list will be zero. Once customers are added, the list will show their ***custId*** and ***numBurritos.*** Using the same strategy (printing contents of the list before and after adding customers and sorting), we can check that the list is being sorted in order of smallest number of burritos first. A list that is being sorted will first show the contents in order of

arrival but after sorting, the customer with the smallest number of burritos will be in the first position and so on.

- The getter and setter methods can be tested by printing the variable they are getting. For example, the getCustId() method will return ***custId***. We can print the ***custId*** produced by the getter method and show that it is indeed working well.
- One of the most crucial tests for this program is whether a customer is asked to leave the shop if it is full. In order to test this, 15 or more customers can be added from the text file. Because some of the first customers will have to come back and sit in the waiting area in order to get their full orders, the customers that arrive when the waiting area is full have to leave the shop and keep trying to gain access until a spot opens. This customer then enters the waiting area, sits, and gets his order fulfilled in the same order as the other customers.
- One of the most difficult, but important things to test in this program is that the number of burritos is decreased by three and that if the customers have burritos left, they will need to pay for the ones they bought and come back to wait for another turn. This is one of the last things we must test, as it requires that all other parts are working properly. Because output is created every time a customer enters the shop, gets burritos made by server x, pays for burritos, and leaves the shop, we can use this output to test the algorithm of re-entry. A good test for this is to have customers with orders of more than 3 burritos who need to get three burritos made, pay for the three burritos, leave the shop, and then come back and sit in the waiting area again. Even though paying and leaving the shop before sitting back in the waiting area was not an explicit requirement of the

project, this was the only way could implement it and the output proved helpful for testing purposes. Any remaining challenges will be discussed in the challenges section.

- Another important part to test is whether the next available server is indeed serving the customer with the lowest number of burritos. Because the make_burritos() method was designed to remove the first customer in the linked list by using the pollFirst() method, we can be sure that the first customer will be removed. A great way to know is to trace the output from the make_burritos() method. If implemented correctly, this will show the servers serving the customers with order sizes of 1 first, then 2, and so on. This behavior was observed in this implementation of Burrito Brothers.

4. Challenges Remaining

4.1 Customers with orders greater than three

In the current solution to Burrito brothers, customers who entered with numBurritos greater than 3 will have to come back to the waiting area and wait to get service. These customers were paying, leaving and then returning to the waiting area correctly. However, the code where they are served by the next available server never ran. Instead, the customers went straight from the waiting area to paying and leaving the shop. Despite many efforts to fix this problem, it is persistent in the implementation. In a real world scenario, an update would be pushed to correct this problem as soon as possible.

5. References

Class LinkedList<E>.   Retrieved from

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Dijkstra, E. (1968). Cooperating Sequential Processes. Programming Languages: NATO Advanced Study Institute, 3–112: Academic Press.

Downey, A. B. (2005). The little book of semaphores. *h ttp://greenteapress. com/semaphores*.

Runnable (Java Platform SE 8). (n.d.).

. Retrieved from https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html