Airport Simulation

Queues, Clocks, and Managing Shared Resources

Drew Meseck

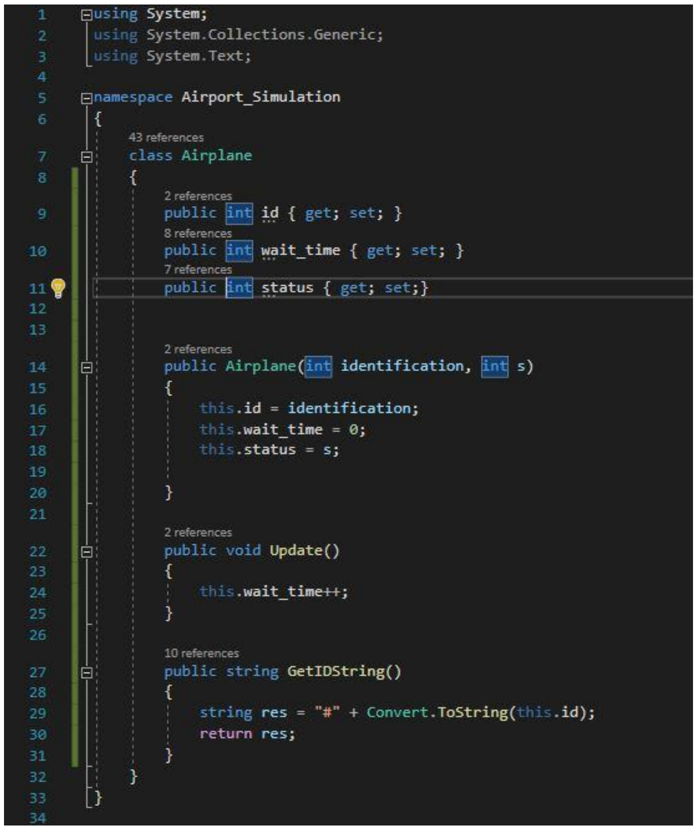
Professor Owrang

10/6/2019

Section 1: Classes

Contained within this total program, and separated into 3 files (program.cs, airport.cs, and airplane.cs) are 3 classes necessary to properly simulate the functions of an airport given the specifications.

Airplane Class:

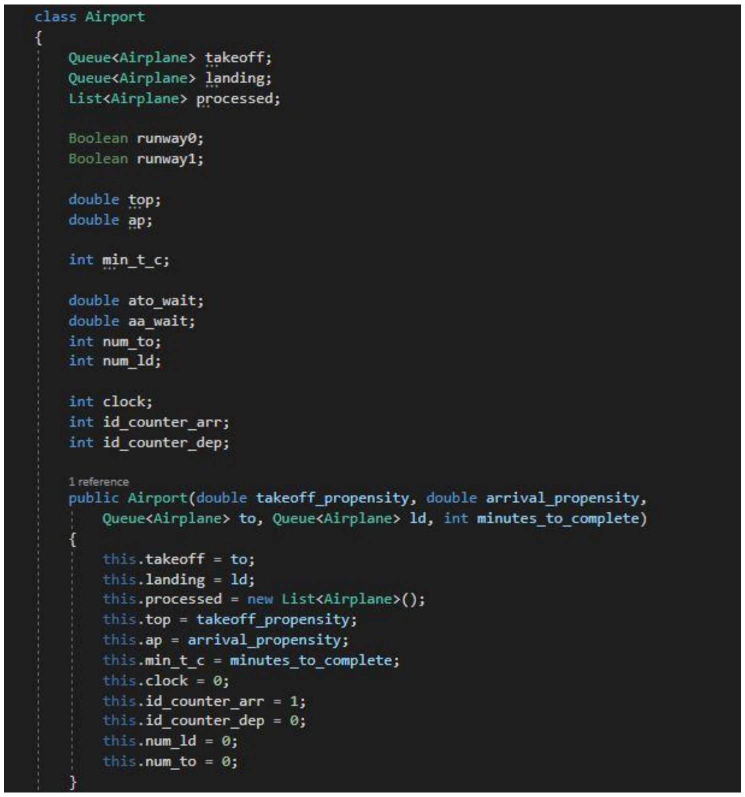


```
using System;
using System.Collections.Generic;
using System.Text;

namespace Airport_Simulation
{
    43 references
    class Airplane
    {
        2 references
        public int id { get; set; }
        8 references
        public int wait_time { get; set; }
        7 references
        public int status { get; set;}


        2 references
        public Airplane(int identification, int s)
        {
            this.id = identification;
            this.wait_time = 0;
            this.status = s;

        }

        2 references
        public void Update()
        {
            this.wait_time++;
        }

        10 references
        public string GetIDString()
        {
            string res = "#" + Convert.ToString(this.id);
            return res;
        }
    }
}
```

This class is designed to give an objective representation of the arrivals and departures, and stores values integral to the statistical output of the program. This is essentially a bin that holds the time any given plane waited in its queue, whether that plane departed or arrived, and the unique ID number assigned to that plane at its creation.

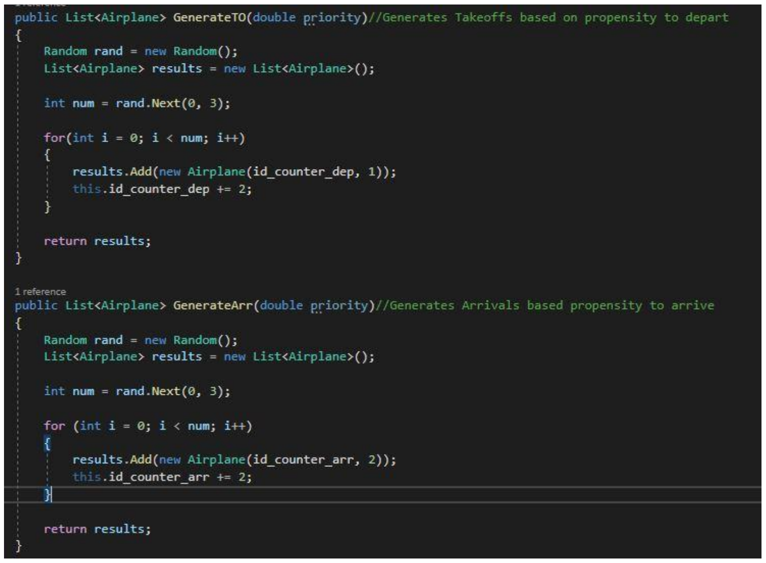
Airport Class:

Nearly the entirety of the program is contained within the airport class. The airport manages everything from the runways, clock, generation of planes, and tracking and aggregating of statistics. This is done largely to simplify the main function in the program class but is also done to better abstract the airport managing its internal systems and resource allocation. In order to break down the individual parts of this class, one must begin by looking at all the airport contains when it is initialized:



```
class Airport
{
    Queue<Airplane> takeoff;
    Queue<Airplane> landing;
    List<Airplane> processed;

    Boolean runway0;
    Boolean runway1;

    double top;
    double ap;

    int min_t_c;

    double ato_wait;
    double aa_wait;
    int num_to;
    int num_ld;

    int clock;
    int id_counter_arr;
    int id_counter_dep;

    1 reference
    public Airport(double takeoff_propensity, double arrival_propensity,
        Queue<Airplane> to, Queue<Airplane> ld, int minutes_to_complete)
    {
        this.takeoff = to;
        this.landing = ld;
        this.processed = new List<Airplane>();
        this.top = takeoff_propensity;
        this.ap = arrival_propensity;
        this.min_t_c = minutes_to_complete;
        this.clock = 0;
        this.id_counter_arr = 1;
        this.id_counter_dep = 0;
        this.num_ld = 0;
        this.num_to = 0;
    }
```

The airport holds each of the Queues needed to schedule the arriving and departing planes (takeoff and landing queues), a list of all processed planes, two Boolean values that track the states of the runways, the defined number of cycles to complete (in 5 minute intervals), priority metrics to define relative priority between takeoffs and landings, a clock, statistical information, and counters for defining the flight information of newly generated planes. This is a lot to keep track of generally, but each of these values are necessary and inherent to the airport aiding in various processes that the airport must complete to function. The first of such processes is generating the planes:

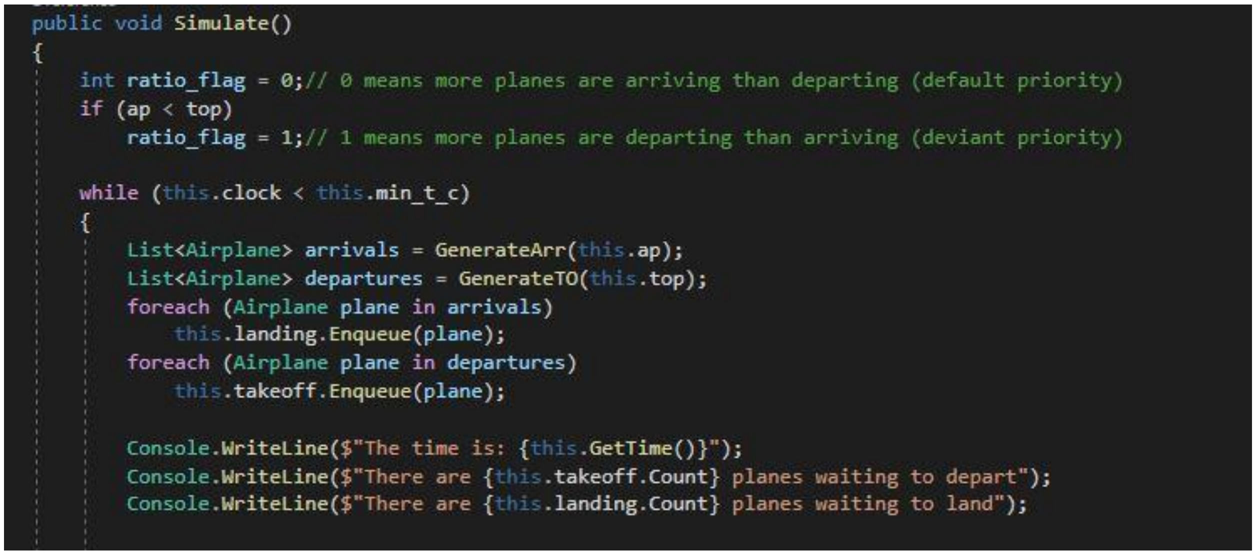


```
public List<Airplane> GenerateTO(double priority)//Generates Takeoffs based on propensity to depart
{
    Random rand = new Random();
    List<Airplane> results = new List<Airplane>();

    int num = rand.Next(0, 3);

    for(int i = 0; i < num; i++)
    {
        results.Add(new Airplane(id_counter_dep, 1));
        this.id_counter_dep += 2;
    }

    return results;
}

1 reference
public List<Airplane> GenerateArr(double priority)//Generates Arrivals based propensity to arrive
{
    Random rand = new Random();
    List<Airplane> results = new List<Airplane>();

    int num = rand.Next(0, 3);

    for (int i = 0; i < num; i++)
    {
        results.Add(new Airplane(id_counter_arr, 2));
        this.id_counter_arr += 2;
    }

    return results;
}
```

Generating the planes is quite simple. By the specifications of the program, at each unit of time 0 to 3 planes may be generated in each queue. When a new plane is generated, they are assigned a takeoff or landing status and a unique ID number that can later be referenced.

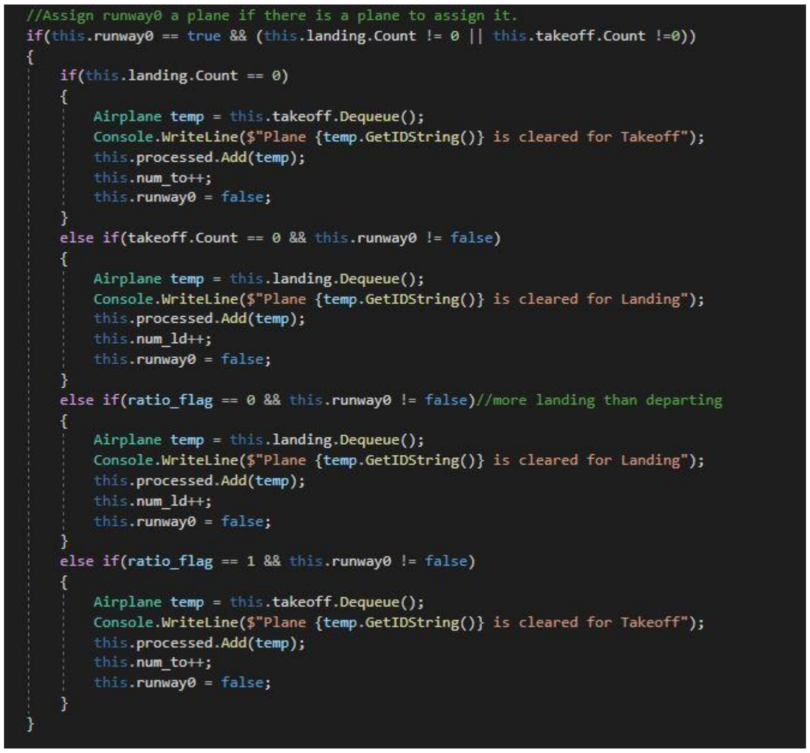
The actual simulation takes place in the Simulate() function, that initializes the clock, and performs all of the printing operations necessary to display the information being created at each step. It also contains the method by which resources are assigned. I approached this problem by assigning a relatively higher priority to landing planes, and a semi-random assignment when both queues contain planes. However, if either queue is empty, the non-empty queue will have planes contained within it assigned to any runway so that no runway is idle unless there are no planes to land (in which case they must be idle).

The Simulation is initialized like this:



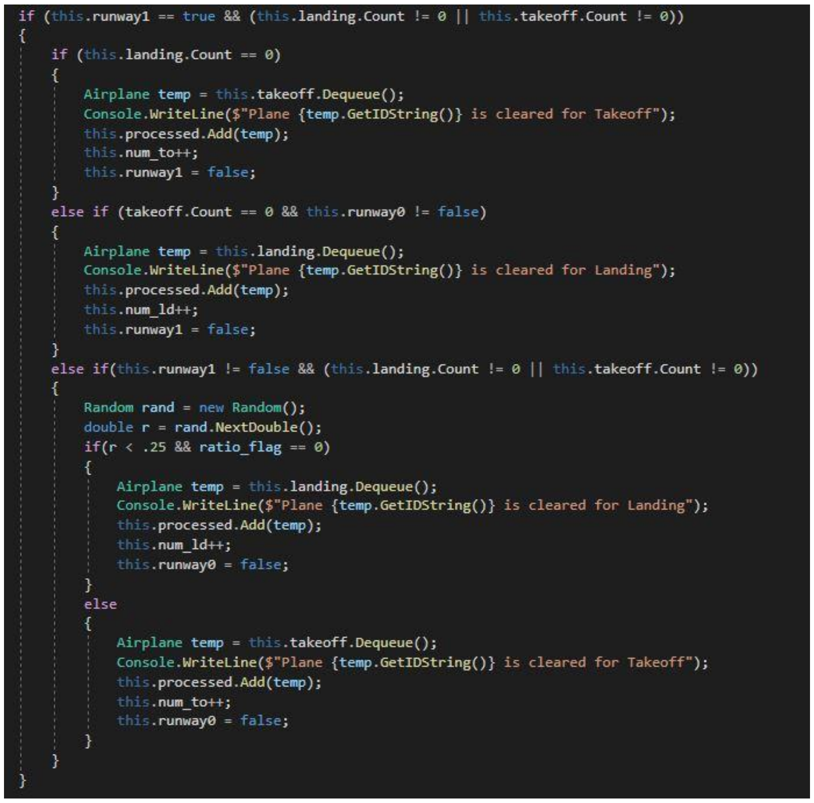
```
public void Simulate()
{
    int ratio_flag = 0;// 0 means more planes are arriving than departing (default priority)
    if (ap < top)
        ratio_flag = 1;// 1 means more planes are departing than arriving (deviant priority)

    while (this.clock < this.min_t_c)
    {
        List<Airplane> arrivals = GenerateArr(this.ap);
        List<Airplane> departures = GenerateTO(this.top);
        foreach (Airplane plane in arrivals)
            this.landing.Enqueue(plane);
        foreach (Airplane plane in departures)
            this.takeoff.Enqueue(plane);

        Console.WriteLine($"The time is: {this.GetTime()}");
        Console.WriteLine($"There are {this.takeoff.Count} planes waiting to depart");
        Console.WriteLine($"There are {this.landing.Count} planes waiting to land");
```

Runway 0 is assigned based on this algorithm:



```
//Assign runway0 a plane if there is a plane to assign it.
if(this.runway0 == true && (this.landing.Count != 0 || this.takeoff.Count !=0))
{
    if(this.landing.Count == 0)
    {
        Airplane temp = this.takeoff.Dequeue();
        Console.WriteLine($"Plane {temp.GetIDString()} is cleared for Takeoff");
        this.processed.Add(temp);
        this.num_to++;
        this.runway0 = false;
    }
    else if(takeoff.Count == 0 && this.runway0 != false)
    {
        Airplane temp = this.landing.Dequeue();
        Console.WriteLine($"Plane {temp.GetIDString()} is cleared for Landing");
        this.processed.Add(temp);
        this.num_ld++;
        this.runway0 = false;
    }
    else if(ratio_flag == 0 && this.runway0 != false)//more landing than departing
    {
        Airplane temp = this.landing.Dequeue();
        Console.WriteLine($"Plane {temp.GetIDString()} is cleared for Landing");
        this.processed.Add(temp);
        this.num_ld++;
        this.runway0 = false;
    }
    else if(ratio_flag == 1 && this.runway0 != false)
    {
        Airplane temp = this.takeoff.Dequeue();
        Console.WriteLine($"Plane {temp.GetIDString()} is cleared for Takeoff");
        this.processed.Add(temp);
        this.num_to++;
        this.runway0 = false;
    }
}
```

Runway 1 is assigned based on this algorithm:



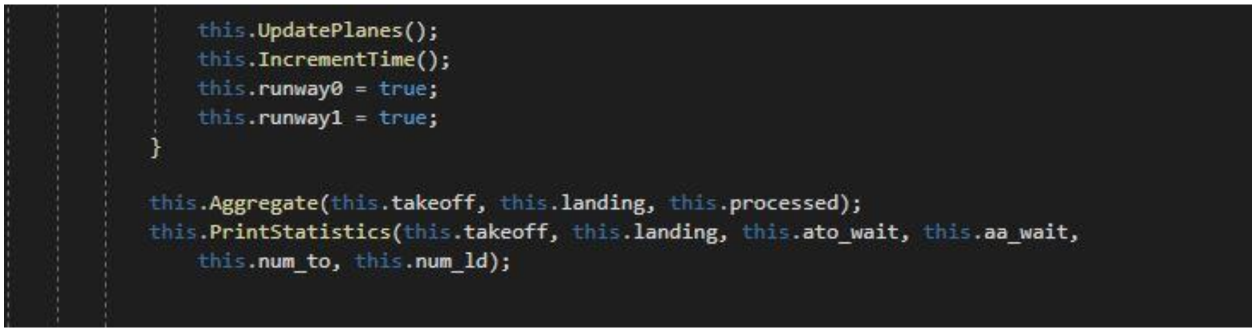
```
if (this.runway1 == true && (this.landing.Count != 0 || this.takeoff.Count != 0))
{
    if (this.landing.Count == 0)
    {
        Airplane temp = this.takeoff.Dequeue();
        Console.WriteLine($"Plane {temp.GetIDString()} is cleared for Takeoff");
        this.processed.Add(temp);
        this.num_to++;
        this.runway1 = false;
    }
    else if (takeoff.Count == 0 && this.runway0 != false)
    {
        Airplane temp = this.landing.Dequeue();
        Console.WriteLine($"Plane {temp.GetIDString()} is cleared for Landing");
        this.processed.Add(temp);
        this.num_ld++;
        this.runway1 = false;
    }
    else if(this.runway1 != false && (this.landing.Count != 0 || this.takeoff.Count != 0))
    {
        Random rand = new Random();
        double r = rand.NextDouble();
        if(r < .25 && ratio_flag == 0)
        {
            Airplane temp = this.landing.Dequeue();
            Console.WriteLine($"Plane {temp.GetIDString()} is cleared for Landing");
            this.processed.Add(temp);
            this.num_ld++;
            this.runway0 = false;
        }
        else
        {
            Airplane temp = this.takeoff.Dequeue();
            Console.WriteLine($"Plane {temp.GetIDString()} is cleared for Takeoff");
            this.processed.Add(temp);
            this.num_to++;
            this.runway0 = false;
        }
    }
}
```

Each cycle, the simulation is updated through 2 functions:

1) Increment Time
2) Update Planes

The Increment time feature simply increments the clock property of the airport so that the time can be updated and so the flag for the maximum number of clock cycles can be evaluated. The Update planes feature updates the status and wait time properties of all planes in both queues.

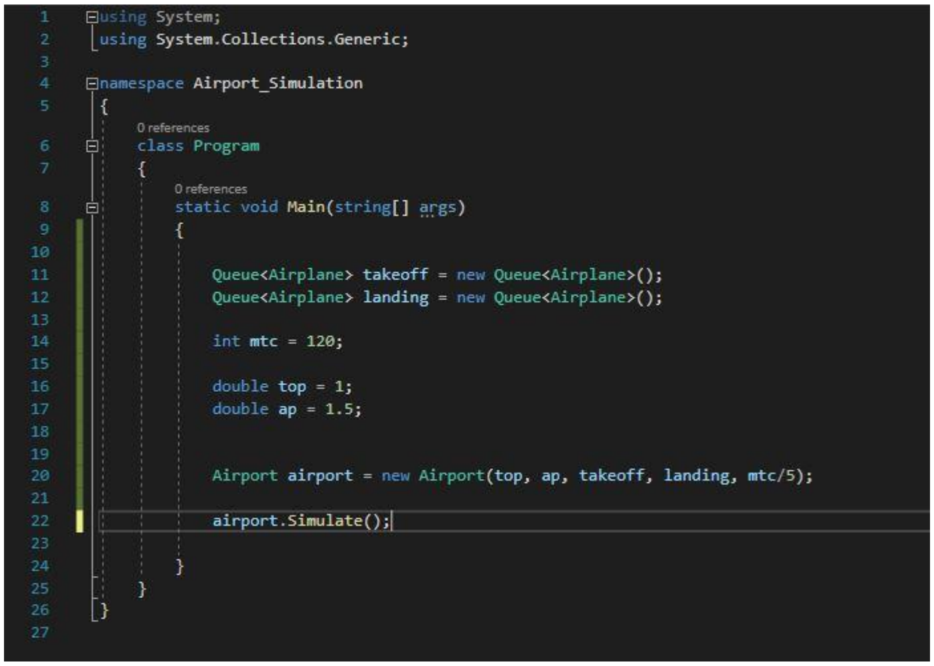
After the maximum clock cycles have been completed, the airport aggregates and prints all the statistics:



```
            this.UpdatePlanes();
            this.IncrementTime();
            this.runway0 = true;
            this.runway1 = true;
        }

        this.Aggregate(this.takeoff, this.landing, this.processed);
        this.PrintStatistics(this.takeoff, this.landing, this.ato_wait, this.aa_wait,
            this.num_to, this.num_ld);
```

Program Class:

The program class solely exists as a control for creating the airport, and simulating the rest of the program:



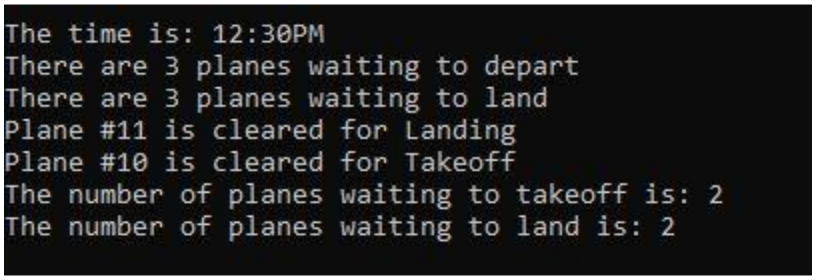
```
using System;
using System.Collections.Generic;

namespace Airport_Simulation
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {

            Queue<Airplane> takeoff = new Queue<Airplane>();
            Queue<Airplane> landing = new Queue<Airplane>();

            int mtc = 120;

            double top = 1;
            double ap = 1.5;


            Airport airport = new Airport(top, ap, takeoff, landing, mtc/5);

            airport.Simulate();

        }
    }
}

```

Section 2: Results

There are two main output sections:

1) Output at each time interval
2) Aggregate statistical output

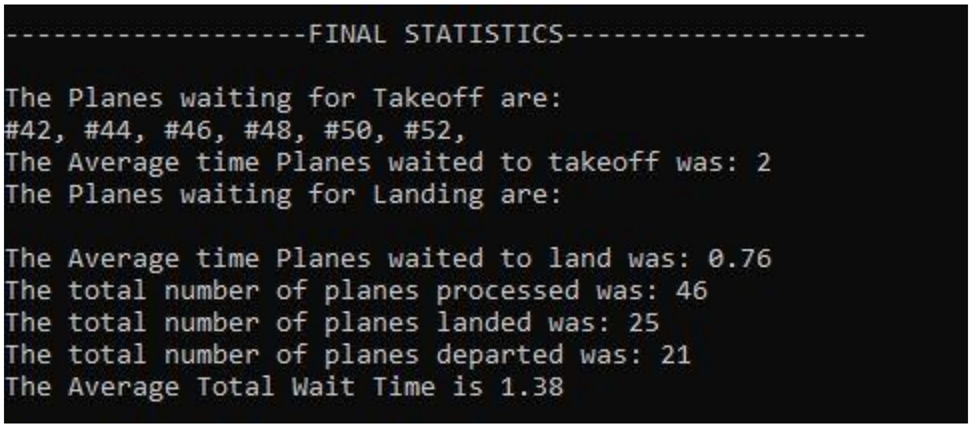
The first of these sections looks like this:



```
The time is: 12:30PM
There are 3 planes waiting to depart
There are 3 planes waiting to land
Plane #11 is cleared for Landing
Plane #10 is cleared for Takeoff
The number of planes waiting to takeoff is: 2
The number of planes waiting to land is: 2
```

Here, essential information about the time, the waiting planes, which planes were cleared for landing or takeoff, and the remaining waiting planes is all displayed. This helps analyze what is happening during the simulation at each step.

The second of these sections looks like this:



```
-------------------FINAL STATISTICS------------------

The Planes waiting for Takeoff are:
#42, #44, #46, #48, #50, #52,
The Average time Planes waited to takeoff was: 2
The Planes waiting for Landing are:

The Average time Planes waited to land was: 0.76
The total number of planes processed was: 46
The total number of planes landed was: 25
The total number of planes departed was: 21
The Average Total Wait Time is 1.38
```

Here, one can analyze the total efficiency of the simulation through various factors, including segmented average wait times, as well as net processed flights. One can also analyze the final components of each queue.