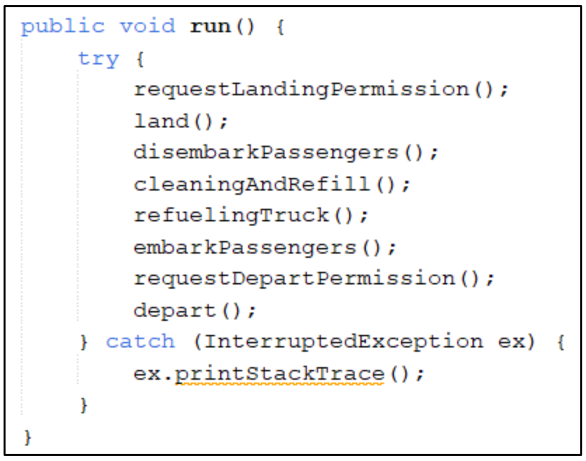
Student Name	:	**Foo Jing Tze**
TP Number	:	**TP066056**
Intake Code	:	**APD2F2209CS(DA)**

Table of Contents

1.0 Basic Requirements

1.1 Plane Flow of Activities

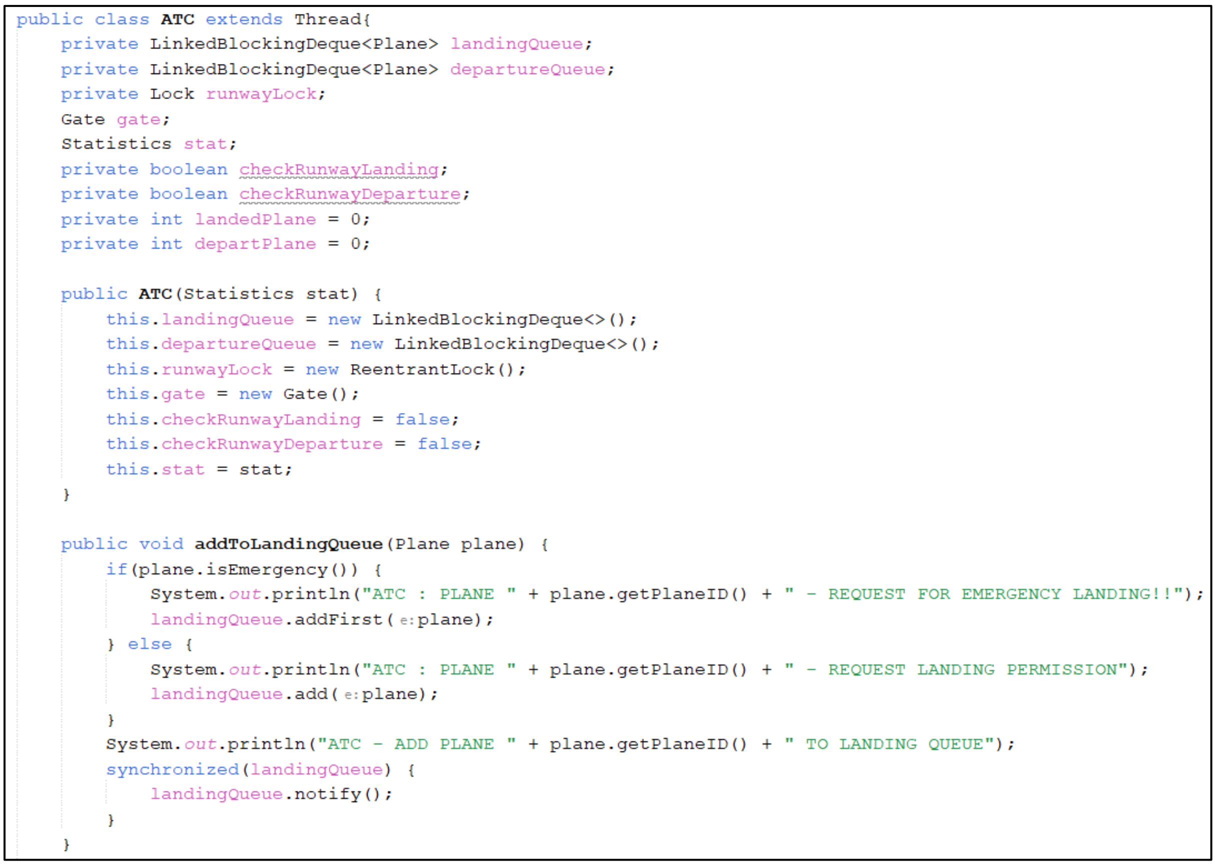


```
public void run() {
    try {
        requestLandingPermission();
        land();
        disembarkPassengers();
        cleaningAndRefill();
        refuelingTruck();
        embarkPassengers();
        requestDepartPermission();
        depart();
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

Figure 1: Flow of Activities in Plane

The **'run()'** method represents the sequence of actions a plane goes through during its cycle. It starts by requesting landing permissions, then lands, disembarks passengers, performs cleaning and refilling tasks, get refueled, embarks passengers, request departure permission and finally departs.

1.2 Plane Request for Landing



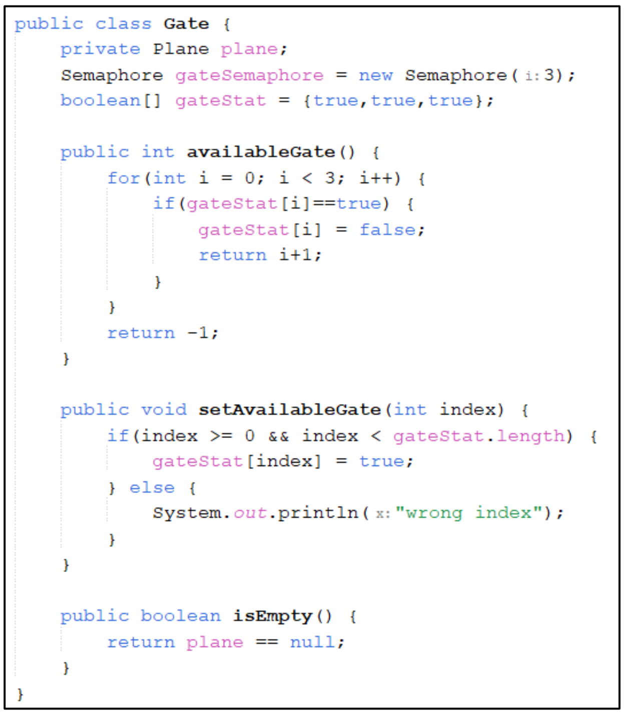
```
public class ATC extends Thread{
    private LinkedBlockingDeque<Plane> landingQueue;
    private LinkedBlockingDeque<Plane> departureQueue;
    private Lock runwayLock;
    Gate gate;
    Statistics stat;
    private boolean checkRunwayLanding;
    private boolean checkRunwayDeparture;
    private int landedPlane = 0;
    private int departPlane = 0;

    public ATC(Statistics stat) {
        this.landingQueue = new LinkedBlockingDeque<>();
        this.departureQueue = new LinkedBlockingDeque<>();
        this.runwayLock = new ReentrantLock();
        this.gate = new Gate();
        this.checkRunwayLanding = false;
        this.checkRunwayDeparture = false;
        this.stat = stat;
    }

    public void addToLandingQueue(Plane plane) {
        if(plane.isEmergency()) {
            System.out.println("ATC : PLANE " + plane.getPlaneID() + " - REQUEST FOR EMERGENCY LANDING!!");
            landingQueue.addFirst( e:plane);
        } else {
            System.out.println("ATC : PLANE " + plane.getPlaneID() + " - REQUEST LANDING PERMISSION");
            landingQueue.add( e:plane);
        }
        System.out.println("ATC - ADD PLANE " + plane.getPlaneID() + " TO LANDING QUEUE");
        synchronized(landingQueue) {
            landingQueue.notify();
        }
    }
```

Figure 2: Code Snippet of Plane Adding into LinkedBlockingDeque named landingQueue

When a plane requests to land, it will be added into a LinkedBlockingDeque named landingQueue. It first checks if the plane is an emergency landing request by calling the **‘isEmergency()’** method on the plane object. If it is an emergency landing, it adds the plane to the front of the landingQueue using **‘addFirst()’** method of **‘LinkedBlockingDeque’** to prioritize the emergency request. If it is not an emergency landing, it adds the plane to the end of the landingQueue using the **‘add()’** method. After that, it notifies any waiting threads that are waiting on the landingQueue using the **‘notify()’** method. This notifies the threads that there is a plane added to the queue, so they can check if it’s their turn to land.

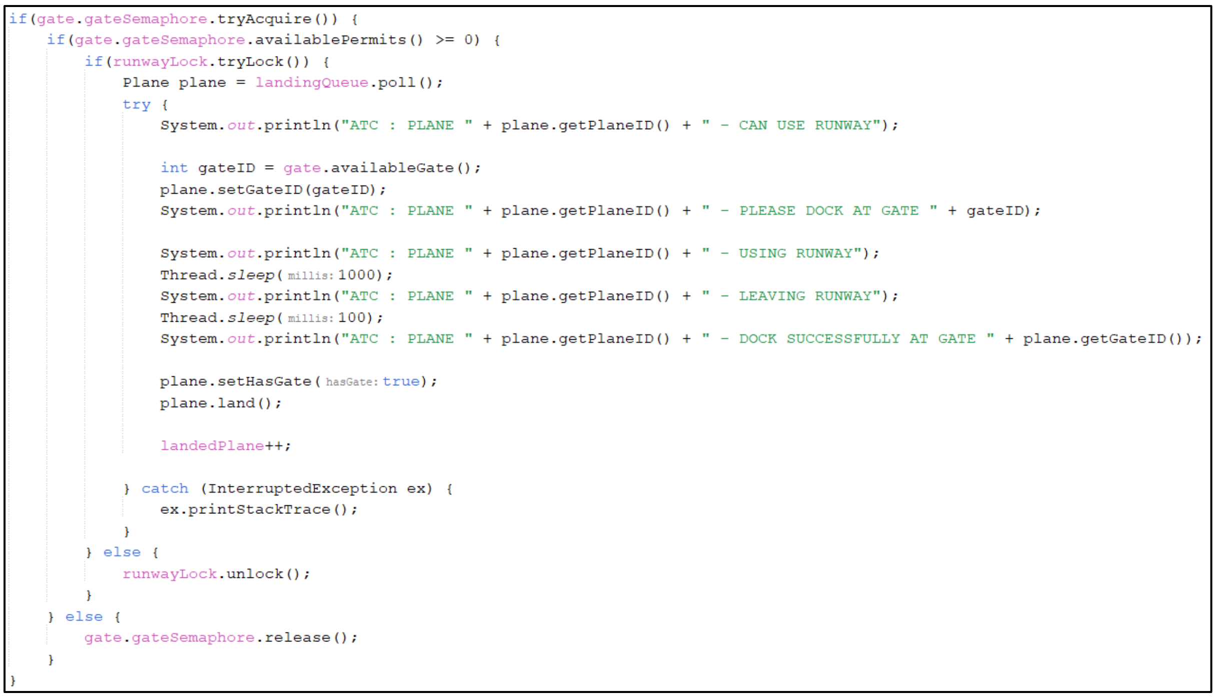
The LinkedBlockingDeque implements the 'BlockingQueue' interface using doubly linked lists. This is used because it enables the addition or removal of parts from either end of the deque, making it suitable for emergency situations. (Prabhu, 2023)

1.3 Runway & Gate



```java
public class Gate {
    private Plane plane;
    Semaphore gateSemaphore = new Semaphore( i: 3);
    boolean[] gateStat = {true,true,true};

    public int availableGate() {
        for(int i = 0; i < 3; i++) {
            if(gateStat[i]==true) {
                gateStat[i] = false;
                return i+1;
            }
        }
        return -1;
    }

    public void setAvailableGate(int index) {
        if(index >= 0 && index < gateStat.length) {
            gateStat[index] = true;
        } else {
            System.out.println( x: "wrong index");
        }
    }

    public boolean isEmpty() {
        return plane == null;
    }
}
```

Figure 3: Gate Class



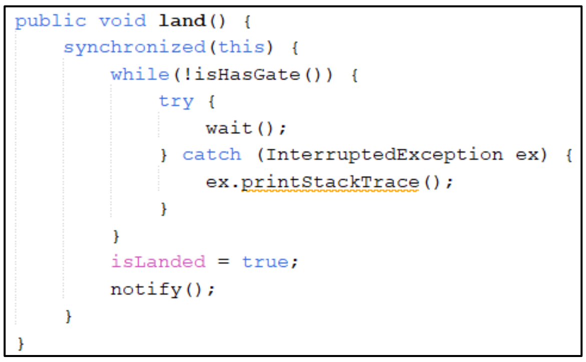
```java
if(gate.gateSemaphore.tryAcquire()) {
    if(gate.gateSemaphore.availablePermits() >= 0) {
        if(runwayLock.tryLock()) {
            Plane plane = landingQueue.poll();
            try {
                System.out.println("ATC : PLANE " + plane.getPlaneID() + " - CAN USE RUNWAY");

                int gateID = gate.availableGate();
                plane.setGateID(gateID);
                System.out.println("ATC : PLANE " + plane.getPlaneID() + " - PLEASE DOCK AT GATE " + gateID);

                System.out.println("ATC : PLANE " + plane.getPlaneID() + " - USING RUNWAY");
                Thread.sleep( millis: 1000);
                System.out.println("ATC : PLANE " + plane.getPlaneID() + " - LEAVING RUNWAY");
                Thread.sleep( millis: 100);
                System.out.println("ATC : PLANE " + plane.getPlaneID() + " - DOCK SUCCESSFULLY AT GATE " + plane.getGateID());

                plane.setHasGate( hasGate: true);
                plane.land();

                landedPlane++;

            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        } else {
            runwayLock.unlock();
        }
    } else {
        gate.gateSemaphore.release();
    }
}
```

Figure 4: Code Snippet for Acquiring Runway and Gate to Land

As shown in Figure 3, Semaphore is applied on gate with an initial count of 3. Semaphore is a type of synchronization that manage access to shared resources. It is a signaling system that enables a constrained number of threads to use a common resource or carry out a specific action simultaneously. (GeeksforGeeks, 2018)

This code block is part of the ATC's run method and handles the process when a plane is ready to land. First, it attempts to acquire a permit from the gateSemaphore, which represents the availability of gates for the plane to dock. If a permit is acquired, it checks if there are still permits remaining to ensure there are no concurrency issues. If there are permits available, it attempts to acquire runwayLock, which represents the runway availability. If the runwayLock is acquired, it retrieves the next plane from the landingQueue and proceeds with landing operations. It assigns the available fate to the plane using the gate's availableGate() method and set the gate ID for the plane. The plane's status is updated, including setting the hasGate flag and invoking the land() method. The landedPlane counter is increased to keep track the number of planes that have successfully landed. If runwayLock cannot be acquired, it releases it to avoid blocking other planes. If there are no permits available, it releases a permit to maintain the balance.

1.4 Landing

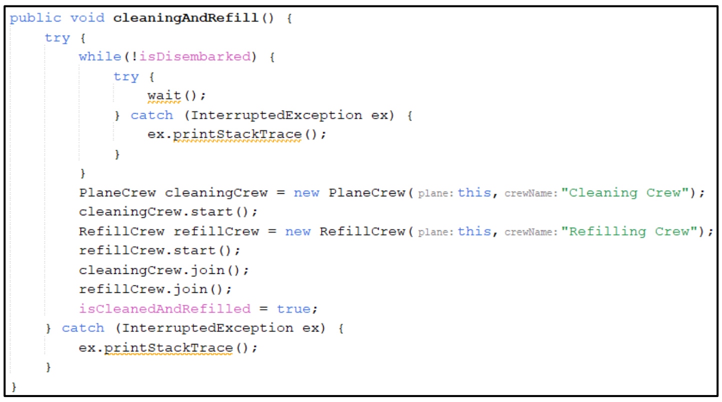


```
public void land() {
    synchronized(this) {
        while(!isHasGate()) {
            try {
                wait();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        isLanded = true;
        notify();
    }
}
```

Figure 5: Code Snippet for Landing

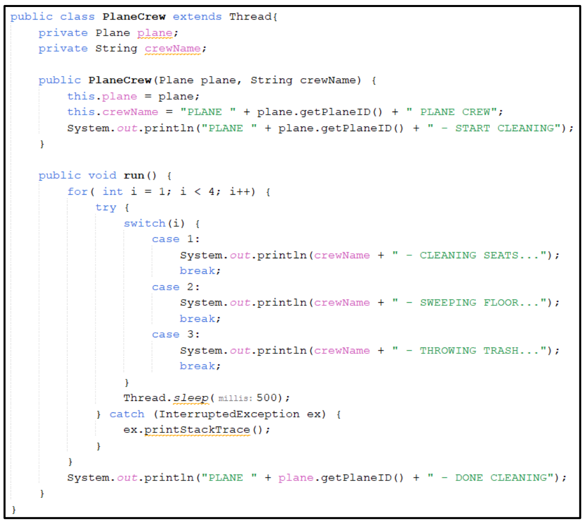
The **'land'** method is synchronized on the current **'Plane**' object. It uses a **'while'** loop to wait until the plane has docked at a gate. If the plane has not yet docked, it waits by calling the **'wait'** method. Once the plane has docked, the **'isLanded'** flag is set to true, and the waiting threads are notified using **'notify()'** method.

1.5 Plane Cleaning and Refill Supplies



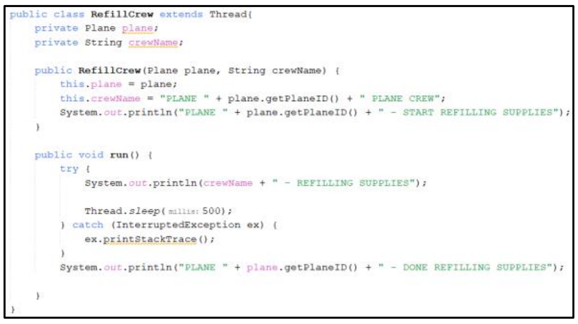
```
public void cleaningAndRefill() {
    try {
        while(!isDisembarked) {
            try {
                wait();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        PlaneCrew cleaningCrew = new PlaneCrew( plane: this, crewName: "Cleaning Crew");
        cleaningCrew.start();
        RefillCrew refillCrew = new RefillCrew( plane: this, crewName: "Refilling Crew");
        refillCrew.start();
        cleaningCrew.join();
        refillCrew.join();
        isCleanedAndRefilled = true;
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

Figure 6: cleaningAndRefill methods in 'Plane' Class



```java
public class PlaneCrew extends Thread{
    private Plane plane;
    private String crewName;

    public PlaneCrew(Plane plane, String crewName) {
        this.plane = plane;
        this.crewName = "PLANE " + plane.getPlaneID() + " PLANE CREW";
        System.out.println("PLANE " + plane.getPlaneID() + " - START CLEANING");
    }

    public void run() {
        for( int i = 1; i < 4; i++) {
            try {
                switch(i) {
                    case 1:
                        System.out.println(crewName + " - CLEANING SEATS...");
                        break;
                    case 2:
                        System.out.println(crewName + " - SWEEPING FLOOR...");
                        break;
                    case 3:
                        System.out.println(crewName + " - THROWING TRASH...");
                        break;
                }
                Thread.sleep( millis: 500);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("PLANE " + plane.getPlaneID() + " - DONE CLEANING");
    }
}
```

Figure 7: Cleaning Crew

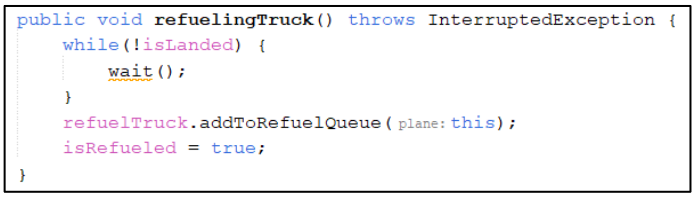


```java
public class RefillCrew extends Thread{
    private Plane plane;
    private String crewName;

    public RefillCrew(Plane plane, String crewName) {
        this.plane = plane;
        this.crewName = "PLANE " + plane.getPlaneID() + " PLANE CREW";
        System.out.println("PLANE " + plane.getPlaneID() + " - START REFILLING SUPPLIES");
    }

    public void run() {
        try {
            System.out.println(crewName + " - REFILLING SUPPLIES");

            Thread.sleep( millis: 500);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println("PLANE " + plane.getPlaneID() + " - DONE REFILLING SUPPLIES");

    }
}
```

Figure 8: Refilling Crew

The **'cleaningAndRefill()'** method ensures that the plane is fully disembarked before initiating the cleaning and refilling process. It creates and start separate threads for the cleaning crew and refill crew, waits for their completion, and marks the plane as cleaned and refilled.

1.6 Plane Refueling

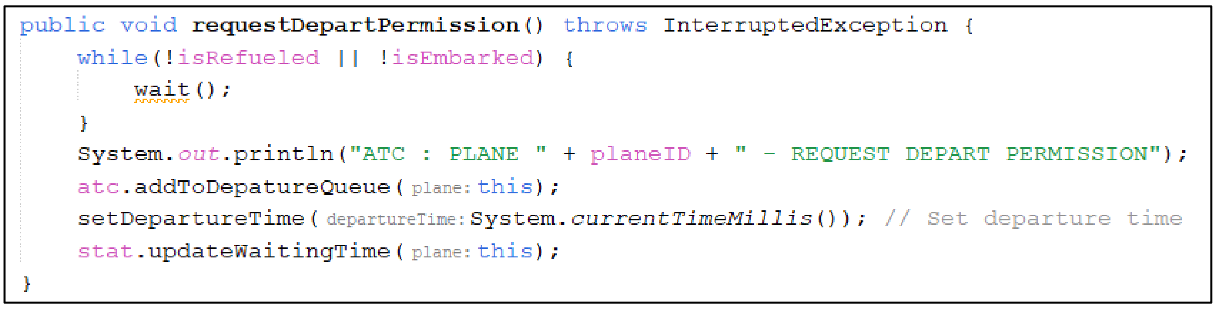


```java
public void refuelingTruck() throws InterruptedException {
    while(!isLanded) {
        wait();
    }
    refuelTruck.addToRefuelQueue( plane: this);
    isRefueled = true;
}
```

Figure 9: refuelingTruck Method

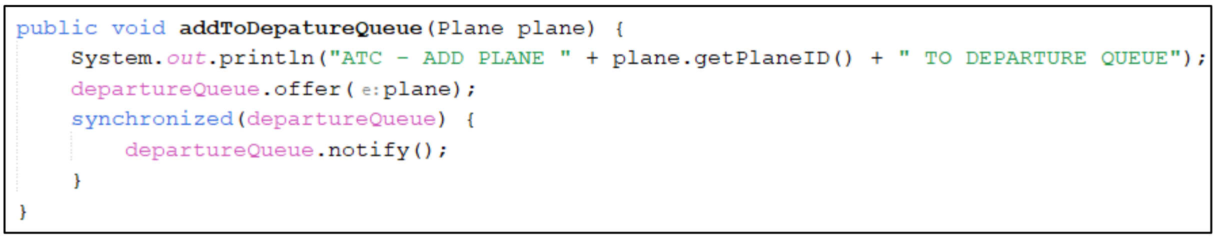
The **'refuelingTruck()'** method waits until the plane has landed by checking the **'isLanded'** flag. Once the plane has landed, it adds itself to the refuelQueue using the 'addToRefuelQueue()' method of the **'refuelTruck'** object. Finally, it sets the **'isRefueled**' flag to true.

1.7 Plane Request for Departure



```java
public void requestDepartPermission() throws InterruptedException {
    while(!isRefueled || !isEmbarked) {
        wait();
    }
    System.out.println("ATC : PLANE " + planeID + " - REQUEST DEPART PERMISSION");
    atc.addToDepatureQueue( plane: this);
    setDepartureTime( departureTime: System.currentTimeMillis()); // Set departure time
    stat.updateWaitingTime( plane: this);
}
```

Figure 10: Code Snippet of Plane Request for Departure



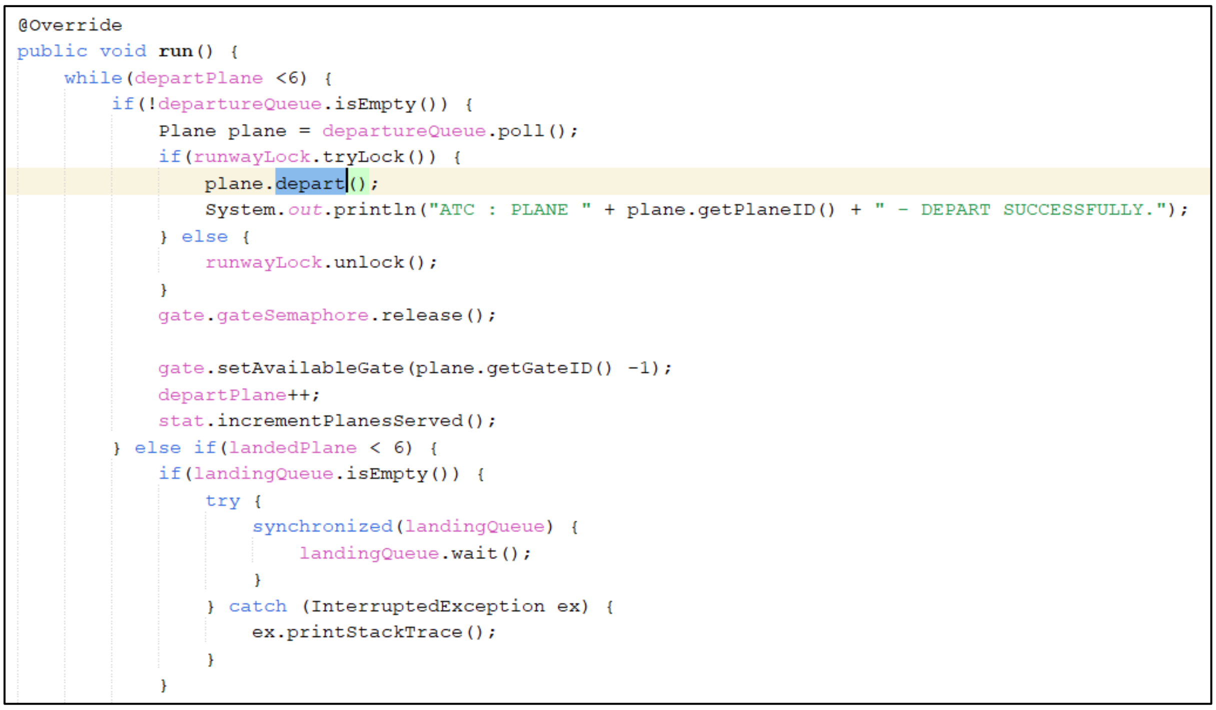
```java
public void addToDepatureQueue(Plane plane) {
    System.out.println("ATC - ADD PLANE " + plane.getPlaneID() + " TO DEPARTURE QUEUE");
    departureQueue.offer( e:plane);
    synchronized(departureQueue) {
        departureQueue.notify();
    }
}
```

Figure 11: Code Snippet of Plane Adding into LinkedBlockingDeque named departureQueue

Based on Figure 10, the **‘requestDepartPermission()’** method waits until the plane is refueled, and passengers are embarked, then adds the plane to the departureQueue and updates the waiting time in the statistics.

Based on Figure 11, the **‘addToDepartureQueue()’** method adds a plane to departure queue and notifies any waiting threads about the new addition.

1.8 Plane Depart



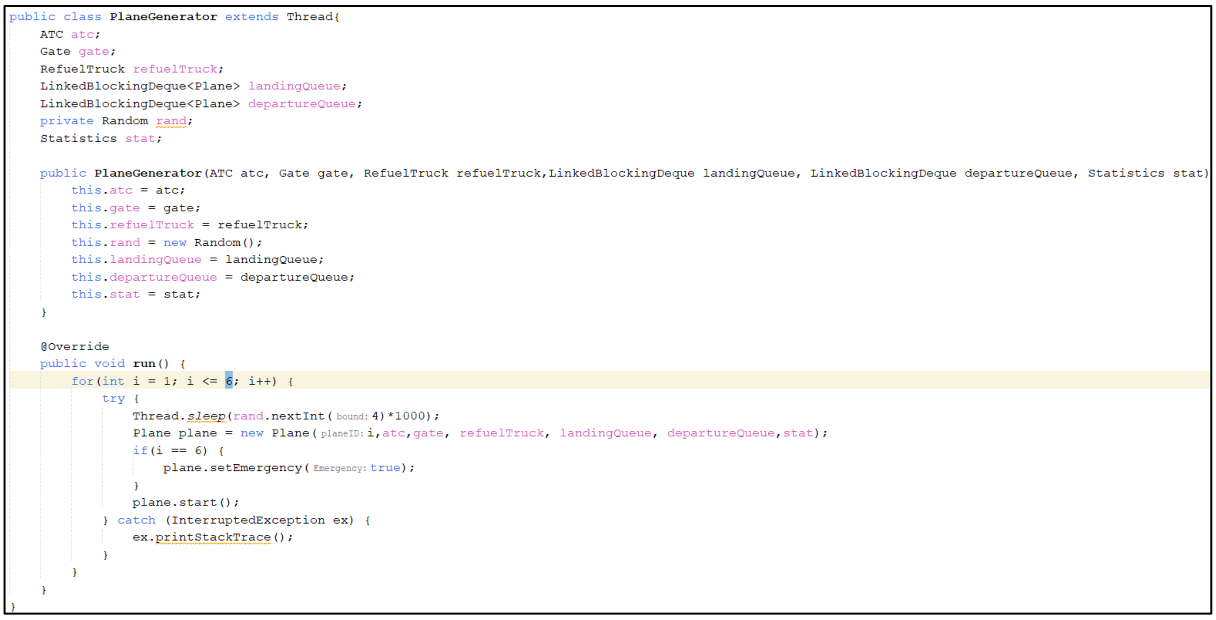
```java
@Override
public void run() {
    while(departPlane <6) {
        if(!departureQueue.isEmpty()) {
            Plane plane = departureQueue.poll();
            if(runwayLock.tryLock()) {
                plane.depart();
                System.out.println("ATC : PLANE " + plane.getPlaneID() + " - DEPART SUCCESSFULLY.");
            } else {
                runwayLock.unlock();
            }
            gate.gateSemaphore.release();

            gate.setAvailableGate(plane.getGateID() -1);
            departPlane++;
            stat.incrementPlanesServed();
        } else if(landedPlane < 6) {
            if(landingQueue.isEmpty()) {
                try {
                    synchronized(landingQueue) {
                        landingQueue.wait();
                    }
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
```

Figure 12: Part of Code Snippet Included Checking of Plane Departed and Landed

In the **‘run()’** method, the ATC continues to process planes until the **‘departPlane’** count reaches 6. If there are planes in the departureQueue, it dequeues a plane and checks if the runway is available. If so, then the plane departs, the runway lock is released, and the gate semaphore is released. The available gate is updated, the **‘departPlane’** count is incremented, and the statistics are updated. If the departureQueue is empty but there are still planes that have landed, the ATC waits for a plane to arrive in the landingQueue or else it will continue processing the remaining code block.

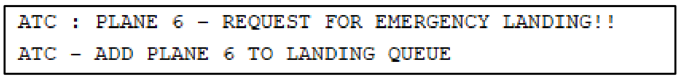
2.0 Additional Requirements

2.1 Emergency Landing



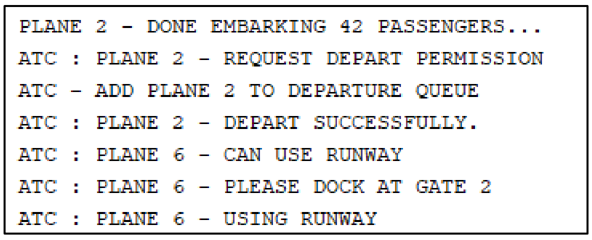
```java
public class PlaneGenerator extends Thread{
    ATC atc;
    Gate gate;
    RefuelTruck refuelTruck;
    LinkedBlockingDeque<Plane> landingQueue;
    LinkedBlockingDeque<Plane> departureQueue;
    private Random rand;
    Statistics stat;

    public PlaneGenerator(ATC atc, Gate gate, RefuelTruck refuelTruck,LinkedBlockingDeque landingQueue, LinkedBlockingDeque departureQueue, Statistics stat)
        this.atc = atc;
        this.gate = gate;
        this.refuelTruck = refuelTruck;
        this.rand = new Random();
        this.landingQueue = landingQueue;
        this.departureQueue = departureQueue;
        this.stat = stat;
    }

    @Override
    public void run() {
        for(int i = 1; i <= 6; i++) {
            try {
                Thread.sleep(rand.nextInt( bound: 4)*1000);
                Plane plane = new Plane( planeID: i,atc,gate, refuelTruck, landingQueue, departureQueue,stat);
                if(i == 6) {
                    plane.setEmergency( Emergency: true);
                }
                plane.start();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Figure 13: Plane Generator



```
ATC : PLANE 6 - REQUEST FOR EMERGENCY LANDING!!
ATC - ADD PLANE 6 TO LANDING QUEUE
```

Figure 14: Sample Output for Plane 6 Generated as Emergency Landing Plane

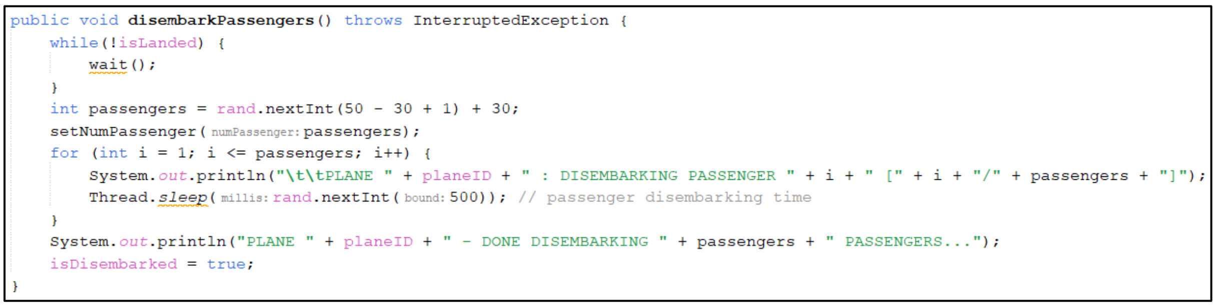


```
PLANE 2 - DONE EMBARKING 42 PASSENGERS...
ATC : PLANE 2 - REQUEST DEPART PERMISSION
ATC - ADD PLANE 2 TO DEPARTURE QUEUE
ATC : PLANE 2 - DEPART SUCCESSFULLY.
ATC : PLANE 6 - CAN USE RUNWAY
ATC : PLANE 6 - PLEASE DOCK AT GATE 2
ATC : PLANE 6 - USING RUNWAY
```

Figure 15: Sample Output for Emergency Landing of Plane 6

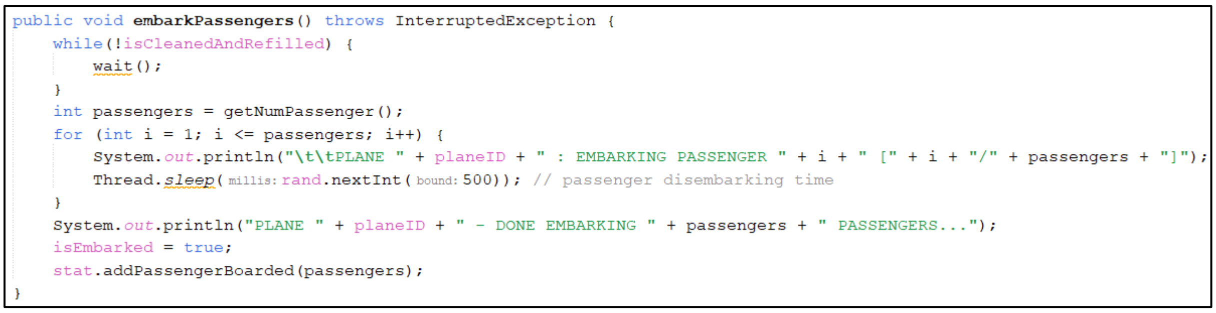
Based on Figure 13, plane 6 is set to Emergency Landing plane. Based on Figure 15, once one plane from the gate departed, plane 6 lands immediately before plane 4 and plane 5 land.

2.2 Disembark and Embark of Passengers



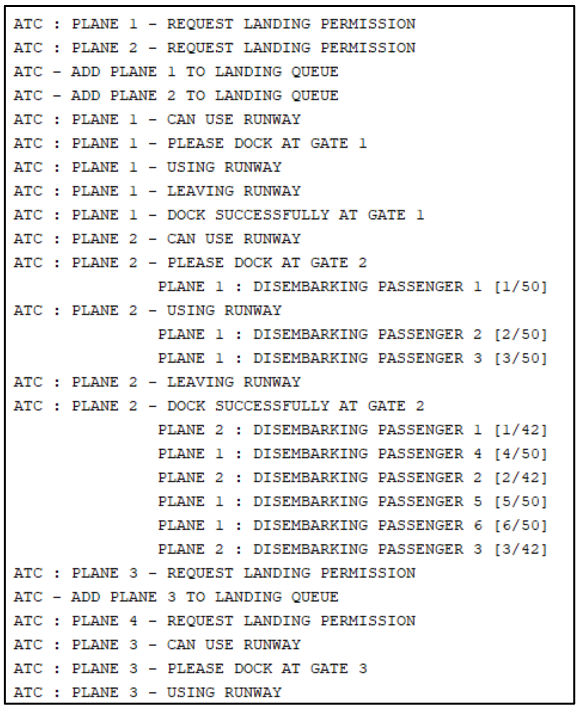
```java
public void disembarkPassengers() throws InterruptedException {
    while(!isLanded) {
        wait();
    }
    int passengers = rand.nextInt(50 - 30 + 1) + 30;
    setNumPassenger( numPassenger: passengers);
    for (int i = 1; i <= passengers; i++) {
        System.out.println("\t\tPLANE " + planeID + " : DISEMBARKING PASSENGER " + i + " [" + i + "/" + passengers + "]");
        Thread.sleep( millis: rand.nextInt( bound: 500)); // passenger disembarking time
    }
    System.out.println("PLANE " + planeID + " - DONE DISEMBARKING " + passengers + " PASSENGERS...");
    isDisembarked = true;
}
```

Figure 16: Disembark Passengers



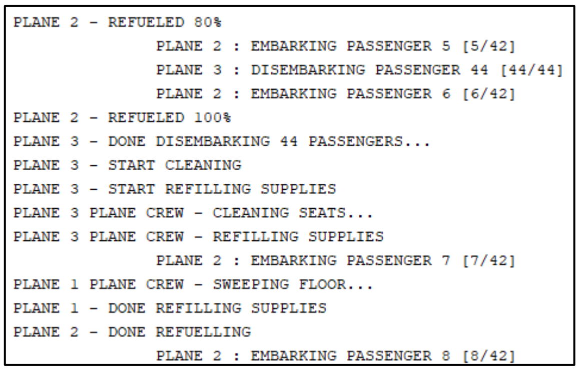
```
public void embarkPassengers() throws InterruptedException {
    while(!isCleanedAndRefilled) {
        wait();
    }
    int passengers = getNumPassenger();
    for (int i = 1; i <= passengers; i++) {
        System.out.println("\t\tPLANE " + planeID + " : EMBARKING PASSENGER " + i + " [" + i + "/" + passengers + "]");
        Thread.sleep( millis: rand.nextInt( bound: 500)); // passenger disembarking time
    }
    System.out.println("PLANE " + planeID + " - DONE EMBARKING " + passengers + " PASSENGERS...");
    isEmbarked = true;
    stat.addPassengerBoarded(passengers);
}
```

Figure 17: Embark Passengers



```
ATC : PLANE 1 - REQUEST LANDING PERMISSION
ATC : PLANE 2 - REQUEST LANDING PERMISSION
ATC - ADD PLANE 1 TO LANDING QUEUE
ATC - ADD PLANE 2 TO LANDING QUEUE
ATC : PLANE 1 - CAN USE RUNWAY
ATC : PLANE 1 - PLEASE DOCK AT GATE 1
ATC : PLANE 1 - USING RUNWAY
ATC : PLANE 1 - LEAVING RUNWAY
ATC : PLANE 1 - DOCK SUCCESSFULLY AT GATE 1
ATC : PLANE 2 - CAN USE RUNWAY
ATC : PLANE 2 - PLEASE DOCK AT GATE 2
                PLANE 1 : DISEMBARKING PASSENGER 1 [1/50]
ATC : PLANE 2 - USING RUNWAY
                PLANE 1 : DISEMBARKING PASSENGER 2 [2/50]
                PLANE 1 : DISEMBARKING PASSENGER 3 [3/50]
ATC : PLANE 2 - LEAVING RUNWAY
ATC : PLANE 2 - DOCK SUCCESSFULLY AT GATE 2
                PLANE 2 : DISEMBARKING PASSENGER 1 [1/42]
                PLANE 1 : DISEMBARKING PASSENGER 4 [4/50]
                PLANE 2 : DISEMBARKING PASSENGER 2 [2/42]
                PLANE 1 : DISEMBARKING PASSENGER 5 [5/50]
                PLANE 1 : DISEMBARKING PASSENGER 6 [6/50]
                PLANE 2 : DISEMBARKING PASSENGER 3 [3/42]
ATC : PLANE 3 - REQUEST LANDING PERMISSION
ATC - ADD PLANE 3 TO LANDING QUEUE
ATC : PLANE 4 - REQUEST LANDING PERMISSION
ATC : PLANE 3 - CAN USE RUNWAY
ATC : PLANE 3 - PLEASE DOCK AT GATE 3
ATC : PLANE 3 - USING RUNWAY
```

Figure 18: Sample Output of Disembarking Passengers Concurrently



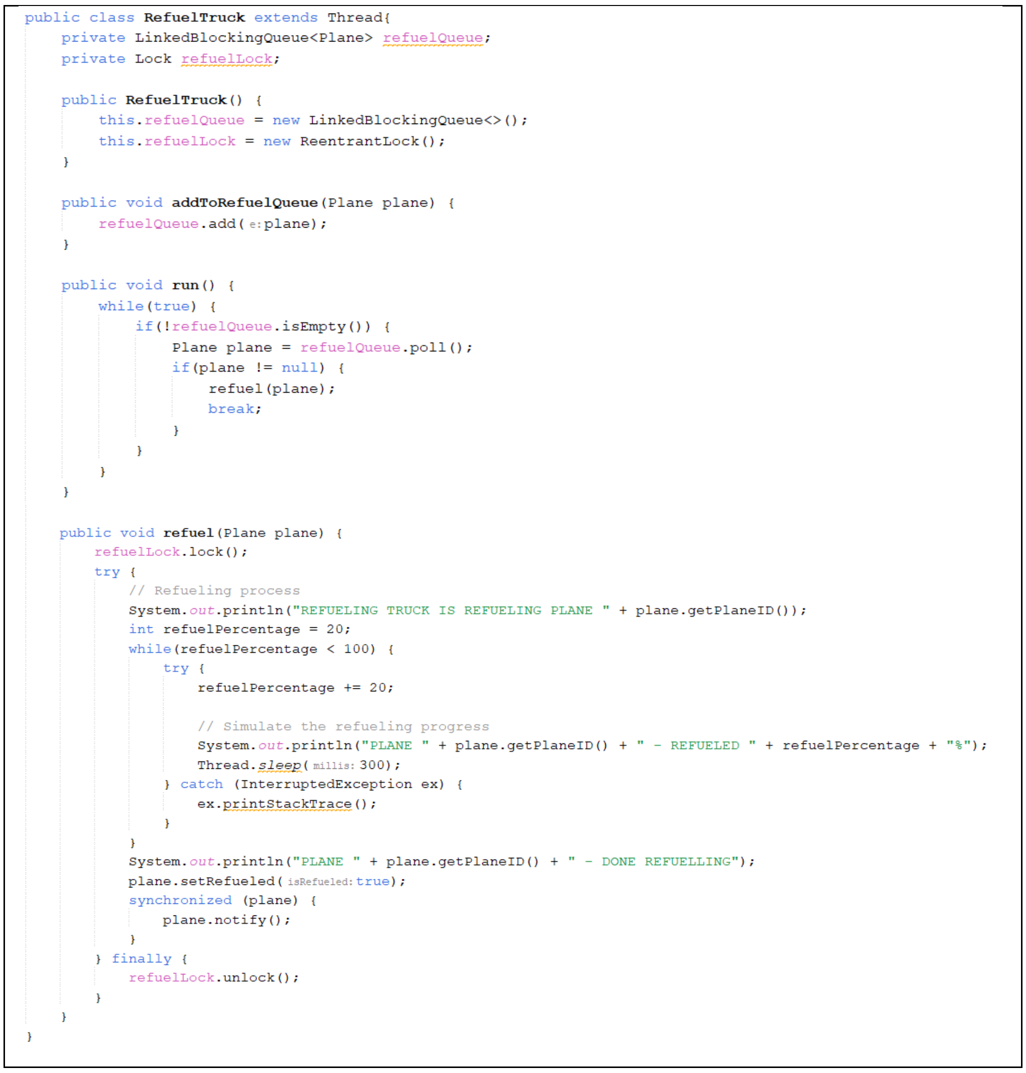
```
PLANE 2 - REFUELED 80%
                PLANE 2 : EMBARKING PASSENGER 5 [5/42]
                PLANE 3 : DISEMBARKING PASSENGER 44 [44/44]
                PLANE 2 : EMBARKING PASSENGER 6 [6/42]
PLANE 2 - REFUELED 100%
PLANE 3 - DONE DISEMBARKING 44 PASSENGERS...
PLANE 3 - START CLEANING
PLANE 3 - START REFILLING SUPPLIES
PLANE 3 PLANE CREW - CLEANING SEATS...
PLANE 3 PLANE CREW - REFILLING SUPPLIES
                PLANE 2 : EMBARKING PASSENGER 7 [7/42]
PLANE 1 PLANE CREW - SWEEPING FLOOR...
PLANE 1 - DONE REFILLING SUPPLIES
PLANE 2 - DONE REFUELLING
                PLANE 2 : EMBARKING PASSENGER 8 [8/42]
```

Figure 19: Sample Output of Embarking Passengers Concurrently

Figure 16 and Figure 17 show disembark and embark of passengers. The **'disembarkPassengers()'** method waits until the plane has landed, then randomly determines the number of passengers and simulates the process of passengers disembarking one by one concurrently with other plane activities. Once all passengers have disembarked, the 'isDisembarked' flag is set to true.

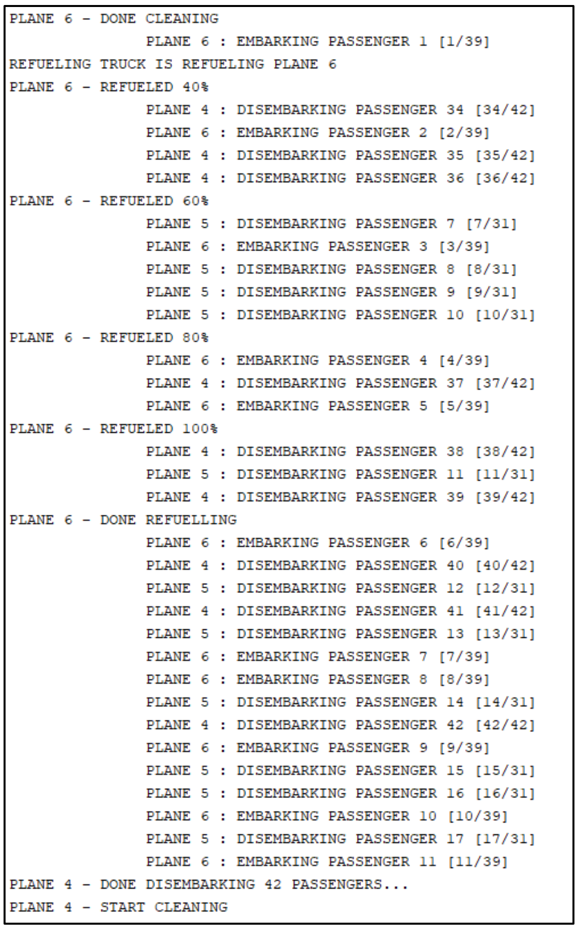
Similarly, the **'embark Passengers()'** method waits until the plane has been cleaned and refilled. It retrieves the number of passengers from the previous disembarking step and simulates the process of passengers boarding one by one concurrently with other plane activities. Once all passengers have boarded, the **'isEmbarked'** glad is set to true, and the passenger count is added to the statistics.

2.3 Refueling Truck



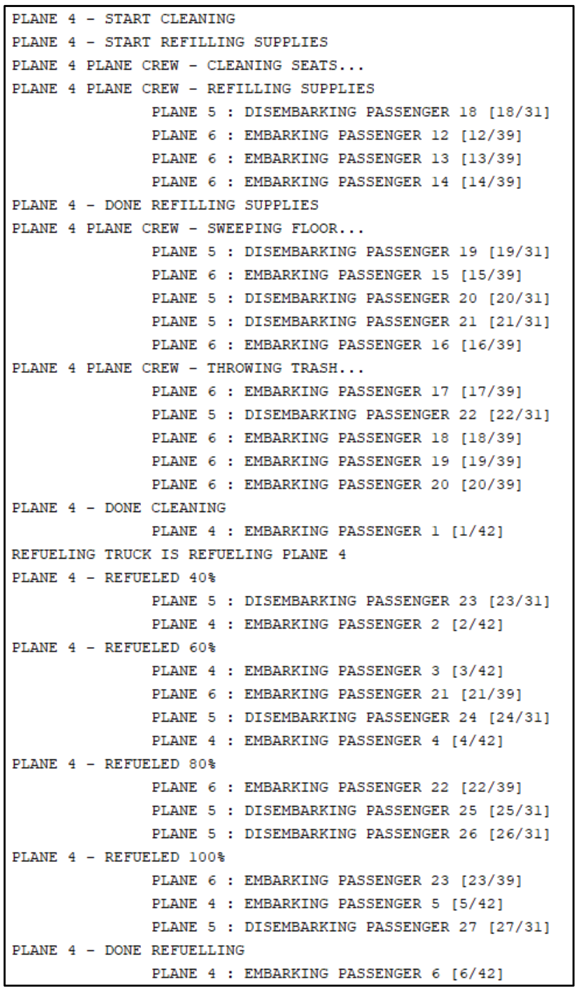
```java
public class RefuelTruck extends Thread{
    private LinkedBlockingQueue<Plane> refuelQueue;
    private Lock refuelLock;

    public RefuelTruck() {
        this.refuelQueue = new LinkedBlockingQueue<>();
        this.refuelLock = new ReentrantLock();
    }

    public void addToRefuelQueue(Plane plane) {
        refuelQueue.add( e: plane);
    }

    public void run() {
        while(true) {
            if(!refuelQueue.isEmpty()) {
                Plane plane = refuelQueue.poll();
                if(plane != null) {
                    refuel(plane);
                    break;
                }
            }
        }
    }

    public void refuel(Plane plane) {
        refuelLock.lock();
        try {
            // Refueling process
            System.out.println("REFUELING TRUCK IS REFUELING PLANE " + plane.getPlaneID());
            int refuelPercentage = 20;
            while(refuelPercentage < 100) {
                try {
                    refuelPercentage += 20;

                    // Simulate the refueling progress
                    System.out.println("PLANE " + plane.getPlaneID() + " - REFUELED " + refuelPercentage + "%");
                    Thread.sleep( millis: 300);
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
            System.out.println("PLANE " + plane.getPlaneID() + " - DONE REFUELLING");
            plane.setRefueled( isRefueled: true);
            synchronized (plane) {
                plane.notify();
            }
        } finally {
            refuelLock.unlock();
        }
    }
}
```

Figure 20 : Refueling Truck Class



```
PLANE 6 - DONE CLEANING
               PLANE 6 : EMBARKING PASSENGER 1 [1/39]
REFUELING TRUCK IS REFUELING PLANE 6
PLANE 6 - REFUELED 40%
               PLANE 4 : DISEMBARKING PASSENGER 34 [34/42]
               PLANE 6 : EMBARKING PASSENGER 2 [2/39]
               PLANE 4 : DISEMBARKING PASSENGER 35 [35/42]
               PLANE 4 : DISEMBARKING PASSENGER 36 [36/42]
PLANE 6 - REFUELED 60%
               PLANE 5 : DISEMBARKING PASSENGER 7 [7/31]
               PLANE 6 : EMBARKING PASSENGER 3 [3/39]
               PLANE 5 : DISEMBARKING PASSENGER 8 [8/31]
               PLANE 5 : DISEMBARKING PASSENGER 9 [9/31]
               PLANE 5 : DISEMBARKING PASSENGER 10 [10/31]
PLANE 6 - REFUELED 80%
               PLANE 6 : EMBARKING PASSENGER 4 [4/39]
               PLANE 4 : DISEMBARKING PASSENGER 37 [37/42]
               PLANE 6 : EMBARKING PASSENGER 5 [5/39]
PLANE 6 - REFUELED 100%
               PLANE 4 : DISEMBARKING PASSENGER 38 [38/42]
               PLANE 5 : DISEMBARKING PASSENGER 11 [11/31]
               PLANE 4 : DISEMBARKING PASSENGER 39 [39/42]
PLANE 6 - DONE REFUELLING
               PLANE 6 : EMBARKING PASSENGER 6 [6/39]
               PLANE 4 : DISEMBARKING PASSENGER 40 [40/42]
               PLANE 5 : DISEMBARKING PASSENGER 12 [12/31]
               PLANE 4 : DISEMBARKING PASSENGER 41 [41/42]
               PLANE 5 : DISEMBARKING PASSENGER 13 [13/31]
               PLANE 6 : EMBARKING PASSENGER 7 [7/39]
               PLANE 6 : EMBARKING PASSENGER 8 [8/39]
               PLANE 5 : DISEMBARKING PASSENGER 14 [14/31]
               PLANE 4 : DISEMBARKING PASSENGER 42 [42/42]
               PLANE 6 : EMBARKING PASSENGER 9 [9/39]
               PLANE 5 : DISEMBARKING PASSENGER 15 [15/31]
               PLANE 5 : DISEMBARKING PASSENGER 16 [16/31]
               PLANE 6 : EMBARKING PASSENGER 10 [10/39]
               PLANE 5 : DISEMBARKING PASSENGER 17 [17/31]
               PLANE 6 : EMBARKING PASSENGER 11 [11/39]
PLANE 4 - DONE DISEMBARKING 42 PASSENGERS...
PLANE 4 - START CLEANING
```

Figure 21: Sample Output for Refueling Truck

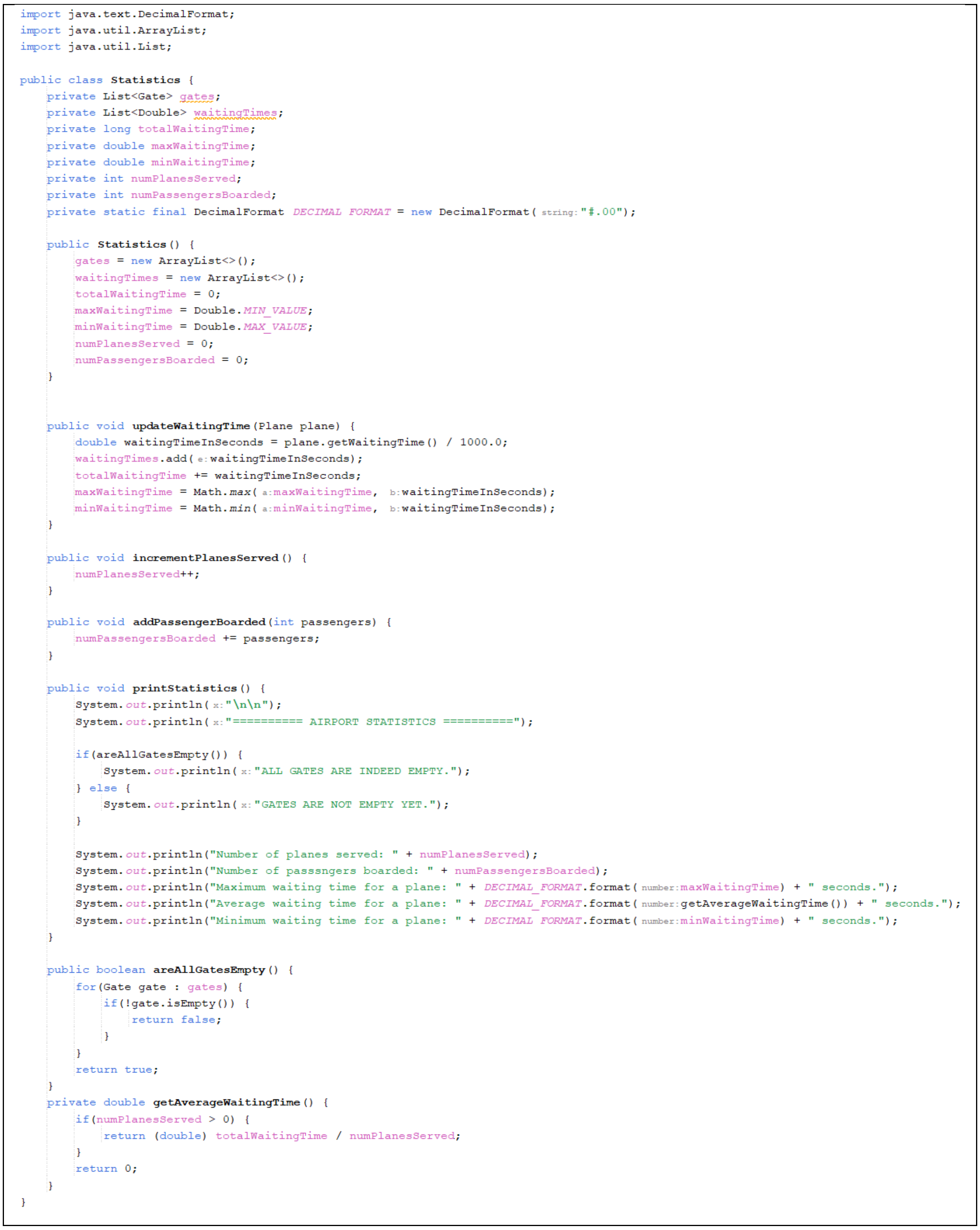


```
PLANE 4 - START CLEANING
PLANE 4 - START REFILLING SUPPLIES
PLANE 4 PLANE CREW - CLEANING SEATS...
PLANE 4 PLANE CREW - REFILLING SUPPLIES
               PLANE 5 : DISEMBARKING PASSENGER 18 [18/31]
               PLANE 6 : EMBARKING PASSENGER 12 [12/39]
               PLANE 6 : EMBARKING PASSENGER 13 [13/39]
               PLANE 6 : EMBARKING PASSENGER 14 [14/39]
PLANE 4 - DONE REFILLING SUPPLIES
PLANE 4 PLANE CREW - SWEEPING FLOOR...
               PLANE 5 : DISEMBARKING PASSENGER 19 [19/31]
               PLANE 6 : EMBARKING PASSENGER 15 [15/39]
               PLANE 5 : DISEMBARKING PASSENGER 20 [20/31]
               PLANE 5 : DISEMBARKING PASSENGER 21 [21/31]
               PLANE 6 : EMBARKING PASSENGER 16 [16/39]
PLANE 4 PLANE CREW - THROWING TRASH...
               PLANE 6 : EMBARKING PASSENGER 17 [17/39]
               PLANE 5 : DISEMBARKING PASSENGER 22 [22/31]
               PLANE 6 : EMBARKING PASSENGER 18 [18/39]
               PLANE 6 : EMBARKING PASSENGER 19 [19/39]
               PLANE 6 : EMBARKING PASSENGER 20 [20/39]
PLANE 4 - DONE CLEANING
               PLANE 4 : EMBARKING PASSENGER 1 [1/42]
REFUELING TRUCK IS REFUELING PLANE 4
PLANE 4 - REFUELED 40%
               PLANE 5 : DISEMBARKING PASSENGER 23 [23/31]
               PLANE 4 : EMBARKING PASSENGER 2 [2/42]
PLANE 4 - REFUELED 60%
               PLANE 4 : EMBARKING PASSENGER 3 [3/42]
               PLANE 6 : EMBARKING PASSENGER 21 [21/39]
               PLANE 5 : DISEMBARKING PASSENGER 24 [24/31]
               PLANE 4 : EMBARKING PASSENGER 4 [4/42]
PLANE 4 - REFUELED 80%
               PLANE 6 : EMBARKING PASSENGER 22 [22/39]
               PLANE 5 : DISEMBARKING PASSENGER 25 [25/31]
               PLANE 5 : DISEMBARKING PASSENGER 26 [26/31]
PLANE 4 - REFUELED 100%
               PLANE 6 : EMBARKING PASSENGER 23 [23/39]
               PLANE 4 : EMBARKING PASSENGER 5 [5/42]
               PLANE 5 : DISEMBARKING PASSENGER 27 [27/31]
PLANE 4 - DONE REFUELLING
               PLANE 4 : EMBARKING PASSENGER 6 [6/42]
```

Figure 22: Sample Output for Refueling Truck (cont'd)

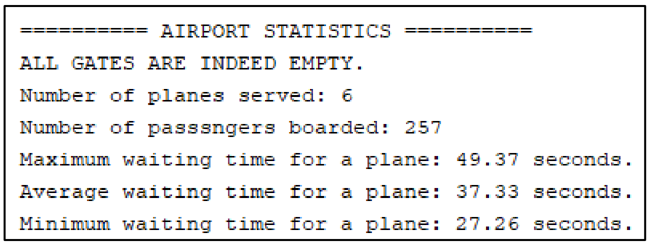
Figure 20 represents the refueling process for a plane using a refueling truck. As there is only one refueling truck, therefore, lock is used to lock the refueling truck when one plane is using the truck. Once the refueling process is completed, the plane object is synchronized, and the waiting threads (if any) are notified. Finally, the refuelLock is released to allow other planes to access the refueling truck. Sample outputs that show only one refueling truck used at one time are shown in Figure 21 and Figure 22.

2.4 Statistics



```
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;

public class Statistics {
    private List<Gate> gates;
    private List<Double> waitingTimes;
    private long totalWaitingTime;
    private double maxWaitingTime;
    private double minWaitingTime;
    private int numPlanesServed;
    private int numPassengersBoarded;
    private static final DecimalFormat DECIMAL_FORMAT = new DecimalFormat( string: "#.00");

    public Statistics() {
        gates = new ArrayList<>();
        waitingTimes = new ArrayList<>();
        totalWaitingTime = 0;
        maxWaitingTime = Double.MIN_VALUE;
        minWaitingTime = Double.MAX_VALUE;
        numPlanesServed = 0;
        numPassengersBoarded = 0;
    }


    public void updateWaitingTime(Plane plane) {
        double waitingTimeInSeconds = plane.getWaitingTime() / 1000.0;
        waitingTimes.add( e: waitingTimeInSeconds);
        totalWaitingTime += waitingTimeInSeconds;
        maxWaitingTime = Math.max( a: maxWaitingTime,  b: waitingTimeInSeconds);
        minWaitingTime = Math.min( a: minWaitingTime,  b: waitingTimeInSeconds);
    }

    public void incrementPlanesServed() {
        numPlanesServed++;
    }

    public void addPassengerBoarded(int passengers) {
        numPassengersBoarded += passengers;
    }

    public void printStatistics() {
        System.out.println( x: "\n\n");
        System.out.println( x: "========== AIRPORT STATISTICS ==========");

        if(areAllGatesEmpty()) {
            System.out.println( x: "ALL GATES ARE INDEED EMPTY.");
        } else {
            System.out.println( x: "GATES ARE NOT EMPTY YET.");
        }

        System.out.println("Number of planes served: " + numPlanesServed);
        System.out.println("Number of passsngers boarded: " + numPassengersBoarded);
        System.out.println("Maximum waiting time for a plane: " + DECIMAL_FORMAT.format( number: maxWaitingTime) + " seconds.");
        System.out.println("Average waiting time for a plane: " + DECIMAL_FORMAT.format( number: getAverageWaitingTime()) + " seconds.");
        System.out.println("Minimum waiting time for a plane: " + DECIMAL_FORMAT.format( number: minWaitingTime) + " seconds.");
    }

    public boolean areAllGatesEmpty() {
        for(Gate gate : gates) {
            if(!gate.isEmpty()) {
                return false;
            }
        }
        return true;
    }
    private double getAverageWaitingTime() {
        if(numPlanesServed > 0) {
            return (double) totalWaitingTime / numPlanesServed;
        }
        return 0;
    }
}
```

Figure 23: Statistics Class



```
========== AIRPORT STATISTICS ==========
ALL GATES ARE INDEED EMPTY.
Number of planes served: 6
Number of passsngers boarded: 257
Maximum waiting time for a plane: 49.37 seconds.
Average waiting time for a plane: 37.33 seconds.
Minimum waiting time for a plane: 27.26 seconds.
```

Figure 24: Sample Output for Statistics

The Statistics class maintains information and calculates statistics related to the airport simulation. It keeps track of gates, number of planes served, number of passengers boarded, waiting times, total waiting time, maximum waiting time, minimum waiting time, and.

The class provides methods to update waiting times, increase the count of planes served, and add the number of passengers boarded. It also has a method to check if all gates are empty.

The printStatistics() method prints out the calculated statistics, including the number of planes served, number of passengers boarded, maximum waiting time, average waiting time, and minimum waiting time.

The class uses a decimal format to display waiting times with two decimal places.

Overall, the Statistics class encapsulates the functionality to track and report statistics related to the airport simulation.

3.0 Requirements Not Met

No requirements that are not met.

4.0 References

Prabhu, R. (2023, March 1). *LinkedBlockingDeque in Java with Examples*. GeeksforGeeks. https://www.geeksforgeeks.org/linkedblockingdeque-in-java-with-examples/

GeeksforGeeks. (2018, December 10). *Semaphore in Java*. GeeksforGeeks. https://www.geeksforgeeks.org/semaphore-in-java/