INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT074-3-2-CCP

CONCURRENT PROGRAMMING

UC2F2102CS(DA)

HAND OUT DATE : 16 AUGUST 2021

HAND IN DATE : 12 NOVEMBER 2021

WEIGHTAGE : 25%

NAME : ONG CHENG KEI

TP NUMBER : TP055620

LECTURER NAME : ZAILAN ARABEE BIN ABDUL SALAM

Table of Contents

BASIC REQUIREMENTS

- **Airplanes can request permission to land and must perform landing afterward**

```
/*
    Method name : requestLanding
    Parameter   : -
    Description : Notify the airport traffic controller to add this airplane to landing queue.
*/
public void requestLanding () {
    String requestString;
    if (hasEmergencies)
        requestString = " : 🟢 Mayday! Mayday! Mayday! Requesting for emergency landing at the airport if possible.";
    else
        requestString = " : Approaching airport in 20 minutes. Requesting permission to land on runway.";
    Thread initialConnection = new Thread ( () -> {
        System.out.println(Thread.currentThread().getName() + requestString);
        controller.addAirplaneToLandingQueue(this);
        System.out.println(Thread.currentThread().getName() + " : Copy that. Joining the landing queue now.");
    },name);
    initialConnection.start();
    startTimer();
}
```

Figure 1 Request Landing method under airplane class

The initial connection is a thread created using a lambda expression, it helps to establish a connection to the airport traffic controller (ATC) under the airplane name.

```
private class Landing implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " : Copy that. " +  Thread.currentThread().getName()+ " is preparing for landing.");
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " crashed unexpectedly.");
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " landed successfully.");
    }
}
```

Figure 2 Landing class (Runnable) under airplane class

Figure 2 shows the landing task is modelled as a class so that it can be passed into a thread and get executed easily.

```java
public Airplane(String name,AirportTrafficController controller,boolean hasEmergencies){
    this.name = name;
    this.controller = controller;
    this.hasEmergencies = hasEmergencies;
    performLanding = new Thread(new Landing(),name);
    performTakeOff = new Thread(new Departure(),name);
    performDocking = new Thread(new Dock(),name);
    performUndocking = new Thread(new Undock(),name);
    performUnloadAndLoad = new Thread(new UnloadAndLoad(),name);
    numberOfPassengersDisembarked = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    numberOfPassengersBoarded = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    requestLanding();
}
```

Figure 3 Airplane constructor

Figure 3 shows performLanding contains the thread that will perform the landing tasks.

```java
// Let airplane perform landing and wait
airplaneToLand.performLanding.start();
try {
    airplaneToLand.performLanding.join();
} catch (InterruptedException error){
    System.out.println("Landing operation interrupted unexpectedly. Check simulation program !!");
    error.printStackTrace();
}
```

Figure 4 Code snippet shows airplane is landing

Figure 4 shows the performLanding thread encapsulated in an airplane instance is started. This implies that the airplane will start to land, and the program will wait for it to finish with the join method.

- **Airplanes must dock at a specific gateway after landed**

```
private class Dock implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is preparing to dock at Gateway : " + assignedGateway);
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " docking process interrupted.");
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " docked successfully at Gateway : " + assignedGateway);
    }
}
```

Figure 5 Dock class (Runnable) under airplane class

```
public Airplane(String name,AirportTrafficController controller,boolean hasEmergencies){
    this.name = name;
    this.controller = controller;
    this.hasEmergencies = hasEmergencies;
    performLanding = new Thread(new Landing(),name);
    performTakeOff = new Thread(new Departure(),name);
    performDocking = new Thread(new Dock(),name);
    performUndocking = new Thread(new Undock(),name);
    performUnloadAndLoad = new Thread(new UnloadAndLoad(),name);
    numberOfPassengersDisembarked = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    numberOfPassengersBoarded = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    requestLanding();
}
```

Figure 6 Airplane constructor

Similarly, the docking task is modelled as a class as shown in Figure 5. Based on Figure 6, performDocking contains the thread that accepts this runnable instance.

```
airplaneToDock.performDocking.start();
try {
    airplaneToDock.performDocking.join();
}catch (InterruptedException e){
    System.out.println("Docking operation interrupted unexpectedly. Check simulation program !!");
    e.printStackTrace();
    continue;
}
```

Figure 7 Code snippet shows airplane is docking

Figure 7 shows the airplane is docking and the program is waiting for the task to finish.

- **Airplanes must unload supplies and passengers before refilling**

```
private class UnloadAndLoad implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is letting passengers to disembark and unloading cargo.");
        try {
            Thread.sleep((long)(numberOfPassengersDisembarked *0.1));
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " unloading operations interrupted.");
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " : A total of " + numberOfPassengersDisembarked + " passengers disembark from " +
                "the airplane, all cargo supplies successfully unloaded.");
        controller.addPassengersArrived(numberOfPassengersDisembarked);
        System.out.println(Thread.currentThread().getName() + " refilling supplies and letting passengers to board the airplane now.");
        try {
            Thread.sleep((long)(numberOfPassengersBoarded *0.1));
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " loading operations interrupted.");
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " : A total of " + numberOfPassengersBoarded + " passengers boarded the airplane, " +
                "all cargo supplies successfully loaded.");
        controller.addPassengersDeparted(numberOfPassengersBoarded);
        requestUndockAndTakeOff();
    }
}
```

Figure 8 Unload and load class (Runnable) under airplane class

Based on Figure 8, the airplanes will always unload the supplies and passengers before letting new passengers come in and load new supplies.

```
public Airplane(String name,AirportTrafficController controller,boolean hasEmergencies){
    this.name = name;
    this.controller = controller;
    this.hasEmergencies = hasEmergencies;
    performLanding = new Thread(new Landing(),name);
    performTakeOff = new Thread(new Departure(),name);
    performDocking = new Thread(new Dock(),name);
    performUndocking = new Thread(new Undock(),name);
    performUnloadAndLoad = new Thread(new UnloadAndLoad(),name);
    numberOfPassengersDisembarked = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    numberOfPassengersBoarded = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    requestLanding();
}
```

Figure 9 Airplane constructor

```
System.out.println(Thread.currentThread().getName()+ " : Noted. Airplane " + airplaneToDock.getName() +
        " can start unload and refill supplies.");
airplaneToDock.performUnloadAndLoad.start();
```

Figure 10 Code snippet shows airplane starts to unload and load

Figure 10 shows the performUnloadAndLoad, a thread that performs the unload and load supplies once the airplane is docked.

- **Airplanes can request to undock and must depart from the airport afterward**

```
/*
    Method name : requestUndockAndTakeOff
    Parameter   : -
    Description : Notify the airport traffic controller to add this airplane to undock queue. This is called after airplane finish loading supplies.
*/
private void requestUndockAndTakeOff () {
    System.out.println(Thread.currentThread().getName() + " : Requesting permission to undock and take off from airport.");
    controller.addAirplaneToUndockQueue(this);
    System.out.println(Thread.currentThread().getName() + " : Copy that. Will wait for further instructions.");

}
```

Figure 11 requestUndockAndTakeOff method under the airplane class

The requestUndockAndTakeOff method is called to notify the ATC that the current airplane is ready to depart from the airport.

```
public Airplane(String name,AirportTrafficController controller,boolean hasEmergencies){
    this.name = name;
    this.controller = controller;
    this.hasEmergencies = hasEmergencies;
    performLanding = new Thread(new Landing(),name);
    performTakeOff = new Thread(new Departure(),name);
    performDocking = new Thread(new Dock(),name);
    performUndocking = new Thread(new Undock(),name);
    performUnloadAndLoad = new Thread(new UnloadAndLoad(),name);
    numberOfPassengersDisembarked = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    numberOfPassengersBoarded = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    requestLanding();
}
```

Figure 12 Airplane constructor

```
private class Undock implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is preparing to undock from Gateway : " + assignedGateway);
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " undocking process interrupted.");
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " undocked successfully from Gateway : " + assignedGateway);
    }
}
```

Figure 13 Undock class (Runnable) under the airplane class

Figure 13 shows the undock task is modelled into a class under the airplane class. The instance of this class will be encapsulated by a thread and it is stored in a variable based on Figure 12.

```
airplaneToUndock.performUndocking.start();
try {
    airplaneToUndock.performUndocking.join();
} catch (InterruptedException e){
    System.out.println("Undocking operation interrupted unexpectedly.");
    e.printStackTrace();
    continue;
}
```

Figure 14 Code snippet shows airplane is undocking

Figure 14 shows the program is waiting for the airplane to finish undocking.

- **Airplanes must be able to depart from the airport**

```
private class Departure implements Runnable {
    @Override
    public void run () {
        System.out.println(Thread.currentThread().getName() + " : Airplane " + Thread.currentThread().getName() +
                " is preparing for takeoff.");
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e){
            System.out.println(Thread.currentThread().getName() + " departure process interrupted.");
        }
        System.out.println(Thread.currentThread().getName() + " : Airplane " + Thread.currentThread().getName() +
                " departed successfully. Thanks.");
    }
}
```

Figure 15 Departure class (Runnable) under the airplane class

Figure 15 shows departure is modelled as a runnable for the airplane. The task will be executed by a thread named performTakeOff shown in Figure 16.

```
public Airplane(String name,AirportTrafficController controller,boolean hasEmergencies){
    this.name = name;
    this.controller = controller;
    this.hasEmergencies = hasEmergencies;
    performLanding = new Thread(new Landing(),name);
    performTakeOff = new Thread(new Departure(),name);
    performDocking = new Thread(new Dock(),name);
    performUndocking = new Thread(new Undock(),name);
    performUnloadAndLoad = new Thread(new UnloadAndLoad(),name);
    numberOfPassengersDisembarked = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    numberOfPassengersBoarded = ThreadLocalRandom.current().nextInt( origin: 25, bound: 51);
    requestLanding();
}
```

Figure 16 Airplane constructor

```
airplaneToDepart.performTakeOff.start();
try {
    airplaneToDepart.performTakeOff.join();
} catch (InterruptedException e){
    System.out.println("Takeoff operation interrupted unexpectedly. Check simulation program!!");
    e.printStackTrace();
    continue;
}
```

Figure 17 Code snippet shows that airplane is taking off

Figure 17 shows the performTakeOff thread is started, and the system will wait for the airplane to successfully depart.

- **Only one airplane is authorized to use the runway at a single point in time**

```java
public class DepartureCoordinator implements Runnable{
    private AirportTrafficController airportTrafficController;

    public DepartureCoordinator(AirportTrafficController airportTrafficController) {
        this.airportTrafficController = airportTrafficController;
    }
    /*
    Method name : run (Method to be called when thread is started)
    Parameter   : Null
    Description : Direct the airplane to depart from runway and record the time taken for the airplane to depart
    Return      : Null
    */
    @Override
    public void run(){
        synchronized (airportTrafficController.runway){
            while (true){
                try {
                    // start off by waiting (will be notified by Airport Traffic Controller shortly)
                    airportTrafficController.runway.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                try {
                    // Throws exception when the queue is empty
                    Airplane airplaneToDepart = airportTrafficController.getDepartureQueue().remove();
                    System.out.println(Thread.currentThread().getName()+ " : Airplane " + airplaneToDepart.getName() + " has the permission to take off now.");
                    // Let airplane to depart and wait for its operation to finish
                    airplaneToDepart.performTakeOff.start();
                    try {
                        airplaneToDepart.performTakeOff.join();
                    } catch (InterruptedException e){
                        System.out.println("Takeoff operation interrupted unexpectedly. Check simulation program!!");
                        e.printStackTrace();
                        continue;
                    }
```

Figure 18 DepartureCoordinator Class (Runnable Task) [1]

```
                    airplaneToDepart.endTimer();
                    airportTrafficController.addTakeoffTime(airplaneToDepart.getElapsedTime());
                    airplaneToDepart.startTimer();
                    System.out.println(Thread.currentThread().getName()+ " : Airplane " + airplaneToDepart.getName() + " have a safe trip.");
                    airportTrafficController.incrementDepartedAirplane();
                }
                catch (NoSuchElementException ignored){
                    // No airplane waiting to depart, thread can go back to sleep.
                }
                // Wake landing coordinator since runway is free
                airportTrafficController.runway.notify();
                // If below two conditions are satisfied, then this thread will terminate
                if (airportTrafficController.isClosed() && airportTrafficController.getTotalAirplanesInPremise() == 0) {
                    return;
                }
            }
        }
    }
}
```

Figure 19 DepartureCoordinator Class (Runnable Task) [2]

```
public class LandingCoordinator implements Runnable{
    private final AirportTrafficController airportTrafficController;

    public LandingCoordinator (AirportTrafficController airportTrafficController){
        this.airportTrafficController = airportTrafficController;
    }
    /*
    Method name : run (Method to be called when thread is started)
    Parameter   : Null
    Description : Direct the airplane to land at runway and record the time taken for the airplane to land
    Return      : Null
    */
    @Override
    public void run() {
        synchronized (airportTrafficController.runway){
            while (true){
                try {
                    // start off by waiting (will be notified by Airport Traffic Controller shortly)
                    airportTrafficController.runway.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (airportTrafficController.getDockingQueue().size() < 2){
                    try {
                        // Throws an exception if the queue is empty
                        Airplane airplaneToLand = airportTrafficController.getLandingQueue().removeFirst();
                        System.out.println(Thread.currentThread().getName()+ " : Airplane " + airplaneToLand.getName() + " has the permission to land now.");
                        // Let airplane perform landing and wait
                        airplaneToLand.performLanding.start();
                        try {
                            airplaneToLand.performLanding.join();
                        } catch (InterruptedException error){
                            System.out.println("Landing operation interrupted unexpectedly. Check simulation program !!");
                            error.printStackTrace();
                        }
```

Figure 20 LandingCoordinator Class (Runnable Task) [1]

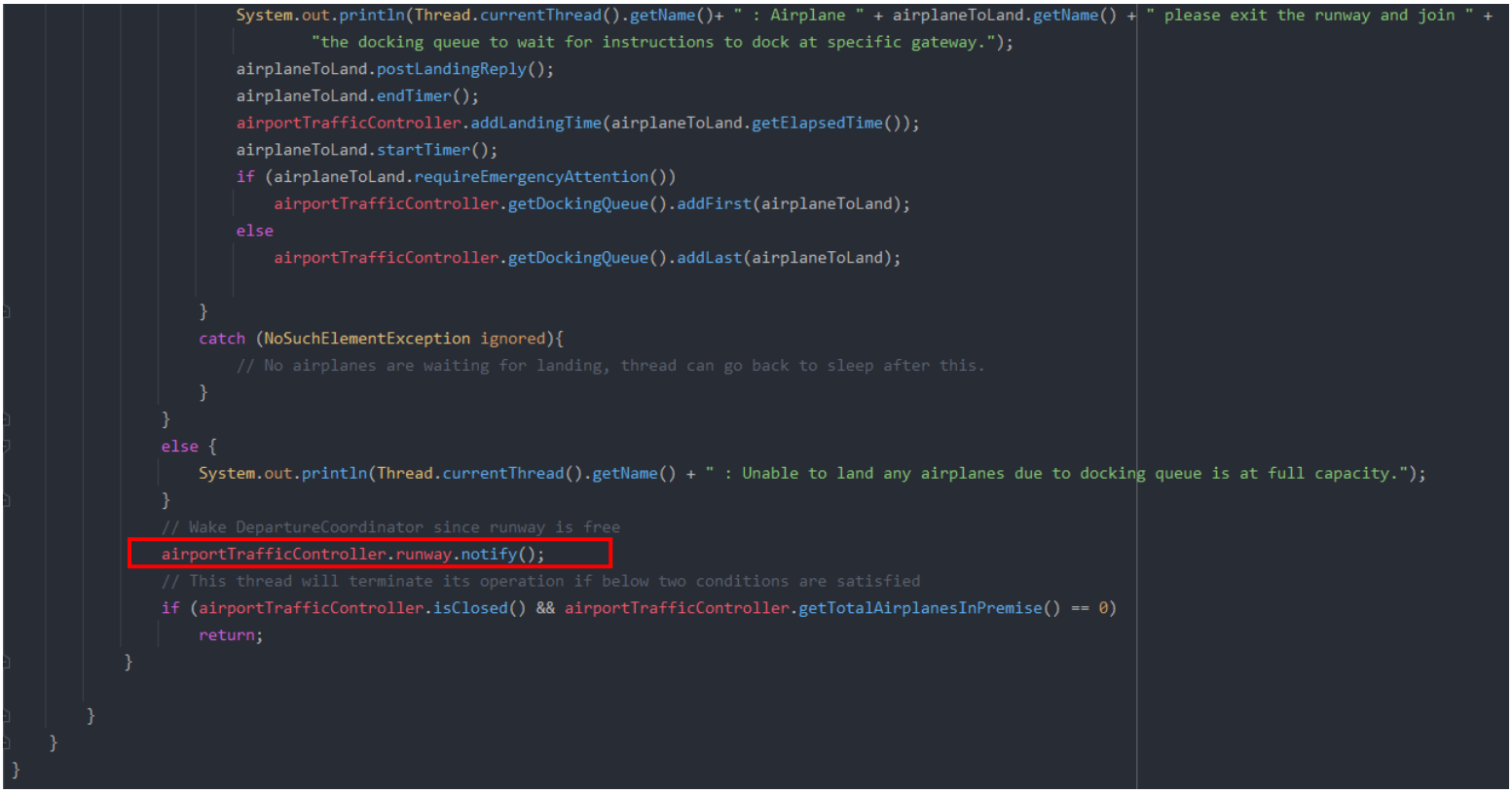


Figure 21 LandingCoordinator Class (Runnable Task) [2]

The departure coordinator thread shares the runway with the landing coordinator thread, hence, the synchronized keyword is used to ensure that only the departure coordinator or landing coordinator gets to run at a single point in time. Each coordinator thread will direct at most one airplane to land or depart after which they will put themselves to sleep and wake the other thread who is waiting on the runway. Besides synchronized, wait and notify methods are used here for both threads to coordinate with each other. Figure 18 and Figure 19 show the entire run method for the departure coordinator class. Figure 20 and Figure 21 show the entire run method for the landing coordinator.

- **Only one airplane is authorized to dock or undock at a single point in time**

```java
public class DockingCoordinator implements Runnable{
    private final AirportTrafficController airportTrafficController;

    public DockingCoordinator(AirportTrafficController airportTrafficController){
        this.airportTrafficController = airportTrafficController;
    }
    /*
    Method name : run (Method to be called when thread is started)
    Parameter   : Null
    Description : Direct the airplane to dock at specific available gateway after acquiring the permission
    Return      : Null
    */
    @Override
    public void run() {
        synchronized (airportTrafficController.landTrafficController){
            while (true) {
                try {
                    airportTrafficController.landTrafficController.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (airportTrafficController.availableGateways.tryAcquire()){
                    try {
                        Airplane airplaneToDock = airportTrafficController.getDockingQueue().removeFirst();
                        String gatewayToDock = airportTrafficController.gateways.poll();
                        System.out.println(Thread.currentThread().getName() + " : Airplane " + airplaneToDock.getName() +
                                " has the permission to dock at gateway " + gatewayToDock);
                        airplaneToDock.setAssignedGateway(gatewayToDock);
                        airplaneToDock.performDocking.start();
                        try {
                            airplaneToDock.performDocking.join();
                        }catch (InterruptedException e){
                            System.out.println("Docking operation interrupted unexpectedly. Check simulation program !!");
                            e.printStackTrace();
                            continue;
                        }
```

Figure 22 Docking coordinator class (Runnable Task) [1]

```
                    airplaneToDock.endTimer();
                    airportTrafficController.addDockingTime(airplaneToDock.getElapsedTime());
                    airplaneToDock.startTimer();
                    System.out.println(Thread.currentThread().getName()+ " : Noted. Airplane " + airplaneToDock.getName() +
                            " can start unload and refill supplies.");
                    airplaneToDock.performUnloadAndLoad.start();
                }
                catch (NoSuchElementException ex){
                    // release back the semaphore since it has already acquired a permit but does not have any airplane to dock at the moment
                    airportTrafficController.availableGateways.release();
                }
            } else {
                System.out.println(Thread.currentThread().getName() + " : Cannot dock any airplane due to no available gateways.");
            }

            airportTrafficController.landTrafficController.notify();
            if (airportTrafficController.isClosed() && airportTrafficController.getTotalAirplanesInPremise() == 0)
                return;
        }
    }
}
```

Figure 23 Docking Coordinator Class (Runnable Task) [2]

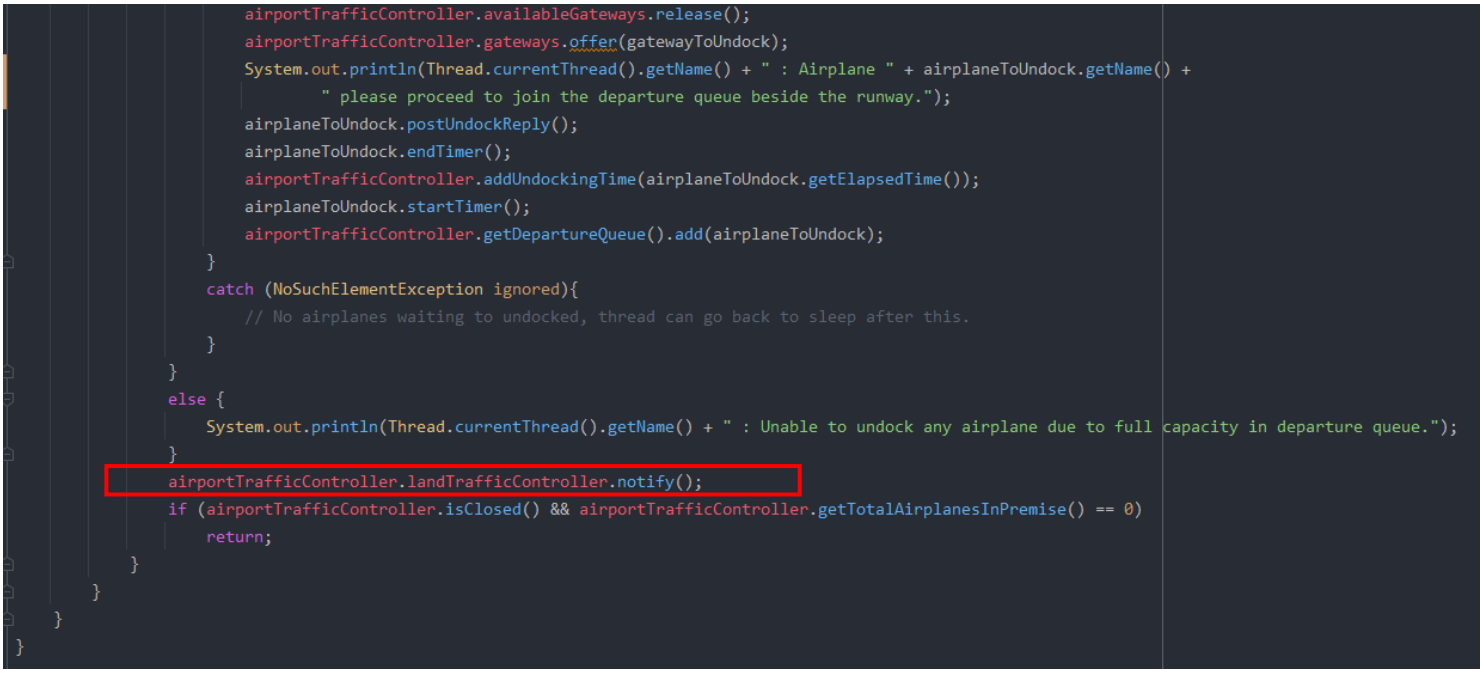
```
public class UndockingCoordinator implements Runnable{
    private final AirportTrafficController airportTrafficController;

    public UndockingCoordinator(AirportTrafficController airportTrafficController) {
        this.airportTrafficController = airportTrafficController;
    }

    /*
    Method name : run (Method to be called when thread is started)
    Parameter   : Null
    Description : Direct the airplane to undock from a specific gateway. Update the time taken, and the status of the gateway to available.
    Return      : Null
    */

    @Override
    public void run() {
        synchronized (airportTrafficController.landTrafficController){
            while (true){
                try {
                    airportTrafficController.landTrafficController.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (airportTrafficController.getDepartureQueue().size() < 2){
                    try {
                        // below throws an exception when the queue is empty
                        Airplane airplaneToUndock = airportTrafficController.getUndockingQueue().remove();
                        String gatewayToUndock = airplaneToUndock.getAssignedGateway();
                        System.out.println(Thread.currentThread().getName() + " : Airplane " + airplaneToUndock.getName() +
                                " has the permission to undock from gateway " + gatewayToUndock);
                        airplaneToUndock.performUndocking.start();
                        try {
                            airplaneToUndock.performUndocking.join();
                        } catch (InterruptedException e){
                            System.out.println("Undocking operation interrupted unexpectedly.");
                            e.printStackTrace();
                            continue;
                        }
```

Figure 24 Undocking Coordinator class (Runnable Task) [1]



```
                    airportTrafficController.availableGateways.release();
                    airportTrafficController.gateways.offer(gatewayToUndock);
                    System.out.println(Thread.currentThread().getName() + " : Airplane " + airplaneToUndock.getName() +
                            " please proceed to join the departure queue beside the runway.");
                    airplaneToUndock.postUndockReply();
                    airplaneToUndock.endTimer();
                    airportTrafficController.addUndockingTime(airplaneToUndock.getElapsedTime());
                    airplaneToUndock.startTimer();
                    airportTrafficController.getDepartureQueue().add(airplaneToUndock);
                }
                catch (NoSuchElementException ignored){
                    // No airplanes waiting to undocked, thread can go back to sleep after this.
                }
            }
            else {
                System.out.println(Thread.currentThread().getName() + " : Unable to undock any airplane due to full capacity in departure queue.");
            }
            airportTrafficController.landTrafficController.notify();
            if (airportTrafficController.isClosed() && airportTrafficController.getTotalAirplanesInPremise() == 0)
                return;
        }
    }
}
```

Figure 25 Undocking Coordinator class (Runnable Task) [2]

Airplanes will have to communicate with the land traffic controller when taxiing around the airport. To minimize the risk of collision, the simulation program will only allow one airplane to dock or undock at a single time. Therefore, the docking coordinator and the undocking coordinator thread will be synchronized on the land traffic controller, so that only one thread will be directing a single airplane to move around the airport. Similarly, the docking coordinator and undocking coordinator will serve up to one airplane when in control before going to sleep and waking the other thread waiting for the synchronized resource. Figure 22 and Figure 23 show the run method for the docking coordinator. Figure 24 and Figure 25 show the run method for the undocking coordinator. Both run methods use synchronized and wait and notify concurrency concepts.

- **Airplanes can only dock at empty gateways assigned by ATC**

```
public AirportTrafficController() {
    runway = new ReentrantLock();
    availableGateways = new Semaphore( permits: 4);
    landTrafficController = new ReentrantLock();
    landingQueue = new LinkedBlockingDeque<>();
    dockingQueue = new LinkedBlockingDeque<>( capacity: 2);
    undockingQueue = new LinkedBlockingQueue<>();
    departureQueue = new LinkedBlockingQueue<>( capacity: 2);
    gateways = new ArrayBlockingQueue<>( capacity: 4,  fair: false, Arrays.asList("1", "2", "3", "4")); // set fairness to false (default)
    closed = false;

    landingThread = new Thread(new LandingCoordinator( airportTrafficController: this),  name: "Air traffic controller");
    dockThread = new Thread(new DockingCoordinator( airportTrafficController: this),  name: "Land traffic Controller");
    undockThread = new Thread(new UndockingCoordinator( airportTrafficController: this),  name: "Land traffic Controller");
    departureThread = new Thread(new DepartureCoordinator( airportTrafficController: this),  name: "Air traffic controller");

    // keep track of the airplanes entered and left the airport
    totalAirplanesDeparted = new AtomicInteger( initialValue: 0);
    totalAirplanesArrived = new AtomicInteger( initialValue: 0);

    minTotalMaxLandingTime = new long[]{Long.MAX_VALUE, 0, Long.MIN_VALUE};
    minTotalMaxDockingTime = new long[]{Long.MAX_VALUE, 0, Long.MIN_VALUE};
    minTotalMaxUndockingTime = new long[]{Long.MAX_VALUE, 0, Long.MIN_VALUE};
    minTotalMaxTakeoffTime = new long[]{Long.MAX_VALUE, 0, Long.MIN_VALUE};

}
```

Figure 26 Airport Traffic Controller constructor

The airport traffic controller will be using a semaphore with 4 permits to keep track of the number of available gateways. Whenever an airplane needs to dock, a permit is acquired, while a permit is returned once the airplane undocks from the gateway. To keep track of the name of the gateways assigned, a simple queue that stores the gateway name is used. All the implementations are shown in Figure 26.

```java
public class DockingCoordinator implements Runnable{
    private final AirportTrafficController airportTrafficController;

    public DockingCoordinator(AirportTrafficController airportTrafficController){
        this.airportTrafficController = airportTrafficController;
    }
    /*
    Method name : run (Method to be called when thread is started)
    Parameter   : Null
    Description : Direct the airplane to dock at specific available gateway after acquiring the permission
    Return      : Null
    */
    @Override
    public void run() {
        synchronized (airportTrafficController.landTrafficController){
            while (true) {
                try {
                    airportTrafficController.landTrafficController.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (airportTrafficController.availableGateways.tryAcquire()){
                    try {
                        Airplane airplaneToDock = airportTrafficController.getDockingQueue().removeFirst();
                        String gatewayToDock = airportTrafficController.gateways.poll();
                        System.out.println(Thread.currentThread().getName() + " : Airplane " + airplaneToDock.getName() +
                                " has the permission to dock at gateway " + gatewayToDock);
                        airplaneToDock.setAssignedGateway(gatewayToDock);
                        airplaneToDock.performDocking.start();
                        try {
                            airplaneToDock.performDocking.join();
                        }catch (InterruptedException e){
                            System.out.println("Docking operation interrupted unexpectedly. Check simulation program !!");
                            e.printStackTrace();
                            continue;
                        }
```

Figure 27 Run method from docking coordinator

The docking coordinator will be communicating with the ATC to get an available gateway for airplanes. The tryAcquire method is used to check if there are any vacant gateways. If there are empty gateways, the name of an empty gateway is retrieved from the queue that stores the available gateways' names as discussed above. The retrieved gateway name is then assigned to the airplane. Figure 27 shows the implementation details.

- **Gateway is set to empty once the airplane undocked**

```
                airportTrafficController.availableGateways.release();
                airportTrafficController.gateways.offer(gatewayToUndock);
                System.out.println(Thread.currentThread().getName() + " : Airplane " + airplaneToUndock.getName() +
                        " please proceed to join the departure queue beside the runway.");
                airplaneToUndock.postUndockReply();
                airplaneToUndock.endTimer();
                airportTrafficController.addUndockingTime(airplaneToUndock.getElapsedTime());
                airplaneToUndock.startTimer();
                airportTrafficController.getDepartureQueue().add(airplaneToUndock);
            }
            catch (NoSuchElementException ignored){
                // No airplanes waiting to undocked, thread can go back to sleep after this.
            }
        }
        else {
            System.out.println(Thread.currentThread().getName() + " : Unable to undock any airplane due to full capacity in departure queue.");
        }
        airportTrafficController.landTrafficController.notify();
        if (airportTrafficController.isClosed() && airportTrafficController.getTotalAirplanesInPremise() == 0)
            return;
    }
  }
}
```

Figure 28 Run method from undocking coordinator

The undocking coordinator is responsible for setting the gateway status to empty once an airplane has undocked from it. The permit that was acquired by the docking coordinator before docking the airplane, will be released, and the gateway name that was assigned to the airplane will be added back to the empty gateways queue by using the offer method.

ADDITIONAL REQUIREMENTS

- **Airplane with emergencies is given top priority to land and dock**

```
public void addAirplaneToLandingQueue(Airplane airplane) {
    String replyString;
    if (airplane.requireEmergencyAttention())
        replyString = " : Airplane " + airplane.getName() + " has the highest priority to land. Please join the landing queue and fly in circles. " +
                "Will allow you to land as soon as possible once the runway is clear.";
    else
        replyString = " : Airplane " + airplane.getName() + " please join the landing queue and fly in circles. " +
                "Will come back to you when the runway is clear for you.";

    Thread replyThread = new Thread(() -> {
        System.out.println(Thread.currentThread().getName() + replyString);
    }, name: "Airport Traffic Controller");
    replyThread.start();
    try {
        replyThread.join();
    } catch (InterruptedException e) {
        System.out.println("Unexpected interruption occurred.");
        e.printStackTrace();
    }
    totalAirplanesArrived.incrementAndGet();
    if (airplane.requireEmergencyAttention())
        landingQueue.addFirst(airplane);
    else
        landingQueue.add(airplane);
}
```

Figure 29 Add airplane to landing queue method under airport traffic controller class

The ATC will check whether an airplane requires emergency attention when adding an airplane into the landing queue. If an airplane requires emergency attention, then the airplane is added at the front of the landing queue with the highest priority to land instead of at the back. Figure 29 shows the code implementation.

```
                System.out.println(Thread.currentThread().getName()+ " : Airplane " + airplaneToLand.getName() + " please exit the runway and join " +
                        "the docking queue to wait for instructions to dock at specific gateway.");
                airplaneToLand.postLandingReply();
                airplaneToLand.endTimer();
                airportTrafficController.addLandingTime(airplaneToLand.getElapsedTime());
                airplaneToLand.startTimer();
                if (airplaneToLand.requireEmergencyAttention())
                    airportTrafficController.getDockingQueue().addFirst(airplaneToLand);
                else
                    airportTrafficController.getDockingQueue().addLast(airplaneToLand);

            }
            catch (NoSuchElementException ignored){
                // No airplanes are waiting for landing, thread can go back to sleep after this.
            }
        }
        else {
            System.out.println(Thread.currentThread().getName() + " : Unable to land any airplanes due to docking queue is at full capacity.");
        }
        // Wake DepartureCoordinator since runway is free
        airportTrafficController.runway.notify();
        // This thread will terminate its operation if below two conditions are satisfied
        if (airportTrafficController.isClosed() && airportTrafficController.getTotalAirplanesInPremise() == 0)
            return;
    }

}
}
```

Figure 30 Run method of landing coordinator

Once the airplane that requires emergency attention landed, the landing coordinator will also proceed to put the airplane at the front of the docking queue so that the airplane will be given the highest priority to dock. Figure 30 shows the code implementation.

- **Sanity check after the airport is closed**

```
public void generateReport() {
    System.out.println("Total number of airplanes : " + totalAirplanesArrived.get());
    System.out.println("Total number of passengers arrived : " + totalPassengersArrived);
    System.out.println("Total number of passengers departed : " + totalPassengersDeparted);
    if (availableGateways.tryAcquire( permits: 4)) {
        System.out.println("All gateways are empty");
        availableGateways.release( permits: 4);
    } else {
        System.out.println("Some gateways are still occupied !! ");
    }
    System.out.println("\n--------- Landing ---------");
    System.out.println("Minimum time taken for airplane to wait and complete landing : " + minTotalMaxLandingTime[0]);
    System.out.println("Average time taken for airplane to wait and complete landing : " + minTotalMaxLandingTime[1] / totalAirplanesArrived.get());
    System.out.println("Maximum time taken for airplane to wait and complete landing : " + minTotalMaxLandingTime[2]);
    System.out.println("\n--------- Docking ---------");
    System.out.println("Minimum time taken for airplane to wait and complete docking : " + minTotalMaxDockingTime[0]);
    System.out.println("Average time taken for airplane to wait and complete docking : " + minTotalMaxDockingTime[1] / totalAirplanesArrived.get());
    System.out.println("Maximum time taken for airplane to wait and complete docking : " + minTotalMaxDockingTime[2]);
    System.out.println("\n--------- Undocking ---------");
    System.out.println("Minimum time taken for airplane to wait and complete undocking : " + minTotalMaxUndockingTime[0]);
    System.out.println("Average time taken for airplane to wait and complete undocking : " + minTotalMaxUndockingTime[1] / totalAirplanesArrived.get());
    System.out.println("Maximum time taken for airplane to wait and complete undocking : " + minTotalMaxUndockingTime[2]);
    System.out.println("\n--------- Takeoff ---------");
    System.out.println("Minimum time taken for airplane to wait and complete takeoff : " + minTotalMaxTakeoffTime[0]);
    System.out.println("Average time taken for airplane to wait and complete takeoff : " + minTotalMaxTakeoffTime[1] / totalAirplanesArrived.get());
    System.out.println("Maximum time taken for airplane to wait and complete takeoff : " + minTotalMaxTakeoffTime[2]);
}
```

Figure 31 Generate report method under airport traffic controller class

When the airport is closed, a tryAcquire method is used to check whether all four of the gateways are empty. The airport should be empty when it is closed, and there should be no leftover airplanes.

- **Detailed statistics generated at the end of the program**

```
public void generateReport() {
    System.out.println("Total number of airplanes : " + totalAirplanesArrived.get());
    System.out.println("Total number of passengers arrived : " + totalPassengersArrived);
    System.out.println("Total number of passengers departed : " + totalPassengersDeparted);
    if (availableGateways.tryAcquire( permits: 4)) {
        System.out.println("All gateways are empty");
        availableGateways.release( permits: 4);
    } else {
        System.out.println("Some gateways are still occupied !! ");
    }
    System.out.println("\n--------- Landing ---------");
    System.out.println("Minimum time taken for airplane to wait and complete landing : " + minTotalMaxLandingTime[0]);
    System.out.println("Average time taken for airplane to wait and complete landing : " + minTotalMaxLandingTime[1] / totalAirplanesArrived.get());
    System.out.println("Maximum time taken for airplane to wait and complete landing : " + minTotalMaxLandingTime[2]);
    System.out.println("\n--------- Docking ---------");
    System.out.println("Minimum time taken for airplane to wait and complete docking : " + minTotalMaxDockingTime[0]);
    System.out.println("Average time taken for airplane to wait and complete docking : " + minTotalMaxDockingTime[1] / totalAirplanesArrived.get());
    System.out.println("Maximum time taken for airplane to wait and complete docking : " + minTotalMaxDockingTime[2]);
    System.out.println("\n--------- Undocking ---------");
    System.out.println("Minimum time taken for airplane to wait and complete undocking : " + minTotalMaxUndockingTime[0]);
    System.out.println("Average time taken for airplane to wait and complete undocking : " + minTotalMaxUndockingTime[1] / totalAirplanesArrived.get());
    System.out.println("Maximum time taken for airplane to wait and complete undocking : " + minTotalMaxUndockingTime[2]);
    System.out.println("\n--------- Takeoff ---------");
    System.out.println("Minimum time taken for airplane to wait and complete takeoff : " + minTotalMaxTakeoffTime[0]);
    System.out.println("Average time taken for airplane to wait and complete takeoff : " + minTotalMaxTakeoffTime[1] / totalAirplanesArrived.get());
    System.out.println("Maximum time taken for airplane to wait and complete takeoff : " + minTotalMaxTakeoffTime[2]);
}
```

Figure 32 Generate report method under airport traffic controller class

The method in Figure 32 shows relevant statistics such as time taken to land, and the number of airplanes served after the airport is closed.

```
public AirportTrafficController() {
    runway = new ReentrantLock();
    availableGateways = new Semaphore( permits: 4);
    landTrafficController = new ReentrantLock();
    landingQueue = new LinkedBlockingDeque<>();
    dockingQueue = new LinkedBlockingDeque<>( capacity: 2);
    undockingQueue = new LinkedBlockingQueue<>();
    departureQueue = new LinkedBlockingQueue<>( capacity: 2);
    gateways = new ArrayBlockingQueue<>( capacity: 4,  fair: false, Arrays.asList("1", "2", "3", "4")); // set fairness to false (default)
    closed = false;

    landingThread = new Thread(new LandingCoordinator( airportTrafficController: this),  name: "Air traffic controller");
    dockThread = new Thread(new DockingCoordinator( airportTrafficController: this),  name: "Land traffic Controller");
    undockThread = new Thread(new UndockingCoordinator( airportTrafficController: this),  name: "Land traffic Controller");
    departureThread = new Thread(new DepartureCoordinator( airportTrafficController: this),  name: "Air traffic controller");

    // keep track of the airplanes entered and left the airport
    totalAirplanesDeparted = new AtomicInteger( initialValue: 0);
    totalAirplanesArrived = new AtomicInteger( initialValue: 0);

    minTotalMaxLandingTime = new long[]{Long.MAX_VALUE, 0, Long.MIN_VALUE};
    minTotalMaxDockingTime = new long[]{Long.MAX_VALUE, 0, Long.MIN_VALUE};
    minTotalMaxUndockingTime = new long[]{Long.MAX_VALUE, 0, Long.MIN_VALUE};
    minTotalMaxTakeoffTime = new long[]{Long.MAX_VALUE, 0, Long.MIN_VALUE};

}
```

Figure 33 Airport traffic controller constructor

Figure 33 shows the ATC constructor. The ATC uses atomic statements to keep track of the number of airplanes that arrived and departed. In addition, the ATC stores the minimum, total, and maximum amount of time taken to perform landing, docking, undocking, and take-off time for the statistics.

```
public void incrementDepartedAirplane() { totalAirplanesDeparted.incrementAndGet(); }
```

Figure 34 increment departed airplane method

Figure 34 shows the method that contains an atomic statement to increment the atomic integer for total airplanes departed.

```
    airplaneToDepart.endTimer();
    airportTrafficController.addTakeoffTime(airplaneToDepart.getElapsedTime());
    airplaneToDepart.startTimer();
    System.out.println(Thread.currentThread().getName()+ " : Airplane " + airplaneToDepart.getName() + " have a safe trip.");
    airportTrafficController.incrementDepartedAirplane();
}
```

Figure 35 Departure coordinator

```java
public void addAirplaneToLandingQueue(Airplane airplane) {
    String replyString;
    if (airplane.requireEmergencyAttention())
        replyString = " : Airplane " + airplane.getName() + " has the highest priority to land. Please join the landing queue and fly in circles. " +
                "Will allow you to land as soon as possible once the runway is clear.";
    else
        replyString = " : Airplane " + airplane.getName() + " please join the landing queue and fly in circles. " +
                "Will come back to you when the runway is clear for you.";

    Thread replyThread = new Thread(() -> {
        System.out.println(Thread.currentThread().getName() + replyString);
    }, name: "Airport Traffic Controller");
    replyThread.start();
    try {
        replyThread.join();
    } catch (InterruptedException e) {
        System.out.println("Unexpected interruption occurred.");
        e.printStackTrace();
    }
    totalAirplanesArrived.incrementAndGet();
    if (airplane.requireEmergencyAttention())
        landingQueue.addFirst(airplane);
    else
        landingQueue.add(airplane);
}
```

Figure 36 add airplane to landing queue method

Figure 35 and Figure 36 show the count of both atomic integers is incremented when airplanes are added into the landing queue or depart from the airport.

```java
    Method name : addLandingTime
    Parameter   : landing time to be added
    Description : add landing time taken by a newly arrived airplane into the array & update the min/max value (if necessary)
    Return      : Null
*/
public void addLandingTime(long newLandingTime) {
    if (newLandingTime < minTotalMaxLandingTime[0])
        minTotalMaxLandingTime[0] = newLandingTime;
    if (newLandingTime > minTotalMaxLandingTime[2])
        minTotalMaxLandingTime[2] = newLandingTime;
    minTotalMaxLandingTime[1] += newLandingTime;
}

/*
Method name : addDockingTime
Parameter   : docking time to be added
Description : add docking time taken by a newly docked airplane into the array & update the min/max value (if necessary)
Return      : Null
*/
public void addDockingTime(long newDockingTime) {
    if (newDockingTime < minTotalMaxDockingTime[0])
        minTotalMaxDockingTime[0] = newDockingTime;
    if (newDockingTime > minTotalMaxDockingTime[2])
        minTotalMaxDockingTime[2] = newDockingTime;
    minTotalMaxDockingTime[1] += newDockingTime;
}
```

Figure 37 Methods used to record time statistics [1]

```java
/*
Method name : addUndockingTime
Parameter   : undocking time to be added
Description : add undocking time taken by a newly undocked airplane into the array & update the min/max value (if necessary)
Return      : Null
*/
public void addUndockingTime(long newUndockingTime) {
    if (newUndockingTime < minTotalMaxUndockingTime[0])
        minTotalMaxUndockingTime[0] = newUndockingTime;
    if (newUndockingTime > minTotalMaxUndockingTime[2])
        minTotalMaxUndockingTime[2] = newUndockingTime;
    minTotalMaxUndockingTime[1] += newUndockingTime;
}

/*
    Method name : addTakeoffTime
    Parameter   : takeoff time to be added
    Description : add departure time taken by a newly departed airplane into the array & update the min/max value (if necessary)
    Return      : Null
*/

public void addTakeoffTime(long newTakeoffTime) {
    if (newTakeoffTime < minTotalMaxTakeoffTime[0])
        minTotalMaxTakeoffTime[0] = newTakeoffTime;
    if (newTakeoffTime > minTotalMaxTakeoffTime[2])
        minTotalMaxTakeoffTime[2] = newTakeoffTime;
    minTotalMaxTakeoffTime[1] += newTakeoffTime;
}
```

Figure 38 Methods used to record time statistics [2]

Figure 37 and Figure 38 show the method provided by ATC to record the time taken for four basic tasks.

```java
                System.out.println(Thread.currentThread().getName()+ " : Airplane " + airplaneToLand.getName() + " please exit the runway and join " +
                        "the docking queue to wait for instructions to dock at specific gateway.");
                airplaneToLand.postLandingReply();
                airplaneToLand.endTimer();
                airportTrafficController.addLandingTime(airplaneToLand.getElapsedTime());
                airplaneToLand.startTimer();
                if (airplaneToLand.requireEmergencyAttention())
                    airportTrafficController.getDockingQueue().addFirst(airplaneToLand);
                else
                    airportTrafficController.getDockingQueue().addLast(airplaneToLand);

            }
            catch (NoSuchElementException ignored){
                // No airplanes are waiting for landing, thread can go back to sleep after this.
            }
        }
        else {
            System.out.println(Thread.currentThread().getName() + " : Unable to land any airplanes due to docking queue is at full capacity.");
        }
        // Wake DepartureCoordinator since runway is free
        airportTrafficController.runway.notify();
        // This thread will terminate its operation if below two conditions are satisfied
        if (airportTrafficController.isClosed() && airportTrafficController.getTotalAirplanesInPremise() == 0)
            return;
    }

}
}
}
```

Figure 39 Landing coordinator recording time taken to land and reset the timer for docking

```
                    airplaneToDock.endTimer();
                    airportTrafficController.addDockingTime(airplaneToDock.getElapsedTime());
                    airplaneToDock.startTimer();
                    System.out.println(Thread.currentThread().getName()+ " : Noted. Airplane " + airplaneToDock.getName() +
                            " can start unload and refill supplies.");
                    airplaneToDock.performUnloadAndLoad.start();
                }
                catch (NoSuchElementException ex){
                    // release back the semaphore since it has already acquired a permit but does not have any airplane to dock at the moment
                    airportTrafficController.availableGateways.release();
                }
            } else {
                System.out.println(Thread.currentThread().getName() + " : Cannot dock any airplane due to no available gateways.");
            }

            airportTrafficController.landTrafficController.notify();
            if (airportTrafficController.isClosed() && airportTrafficController.getTotalAirplanesInPremise() == 0)
                return;
        }
    }
}
```

Figure 40 Docking coordinator recording the time taken to dock and reset the timer for undock

```
                    airportTrafficController.availableGateways.release();
                    airportTrafficController.gateways.offer(gatewayToUndock);
                    System.out.println(Thread.currentThread().getName() + " : Airplane " + airplaneToUndock.getName() +
                            " please proceed to join the departure queue beside the runway.");
                    airplaneToUndock.postUndockReply();
                    airplaneToUndock.endTimer();
                    airportTrafficController.addUndockingTime(airplaneToUndock.getElapsedTime());
                    airplaneToUndock.startTimer();
                    airportTrafficController.getDepartureQueue().add(airplaneToUndock);
                }
                catch (NoSuchElementException ignored){
                    // No airplanes waiting to undocked, thread can go back to sleep after this.
                }
            }
            else {
                System.out.println(Thread.currentThread().getName() + " : Unable to undock any airplane due to full capacity in departure queue.");
            }
            airportTrafficController.landTrafficController.notify();
            if (airportTrafficController.isClosed() && airportTrafficController.getTotalAirplanesInPremise() == 0)
                return;
        }
    }
}
```

Figure 41 Undocking coordinator recording the time taken to undock and reset the timer for departure

```
                airplaneToDepart.endTimer();
                airportTrafficController.addTakeoffTime(airplaneToDepart.getElapsedTime());
                airplaneToDepart.startTimer();
                System.out.println(Thread.currentThread().getName()+ " : Airplane " + airplaneToDepart.getName() + " have a safe trip.");
                airportTrafficController.incrementDepartedAirplane();
            }
            catch (NoSuchElementException ignored){
                // No airplane waiting to depart, thread can go back to sleep.
            }
            // Wake landing coordinator since runway is free
            airportTrafficController.runway.notify();
            // If below two conditions are satisfied, then this thread will terminate
            if (airportTrafficController.isClosed() && airportTrafficController.getTotalAirplanesInPremise() == 0) {
                return;
            }
        }
    }
}
```

Figure 42 Departure coordinator recording the time taken to depart and reset the timer

Figure 39, Figure 40, Figure 41, and Figure 42 show the time taken by airplane is recorded and stored in the arrays under ATC class.

MISSED REQUIREMENTS

Some of the requirements that were not met in this assignment are listed below :

- Emergencies airplanes that require landing cannot force ATC to revoke the permissions given to airplanes to use the runway (Additional requirement).
- Special scenarios such as weather changes that can delay departure are not implemented (Additional requirement).