Carleton
UNIVERSITY

SYSC 4005 A Course Project

Final Report:

Smiths Falls/Montague Airport (SFMA) Simulator

By: Oyindamola Taiwo-Olupeka

101155729

Due Date: April 12, 2024.

ABSTRACT

The final report consolidates all four deliverables, providing a comprehensive overview of the simulation's design, implementation, and analysis. It includes an overview of the simulation model and its components, as outlined in Deliverable 1. Additionally, detailed analysis of the current operational performance, identifying constraints and areas for improvement, is presented in Deliverable 2. Deliverable 3 presents alternative operating policies and their potential impact on performance. Furthermore, Deliverable 4 encompasses the justification and evaluation of the recommended policy, along with conclusions. Finally, the report encompasses the source codes for the simulation's programs providing a technical foundation for understanding the simulation's functionality and the basis for the proposed improvements.

TABLE OF CONTENT

1.0 PROBLEM FORMULATION

The problem we are presented with is to simulate the check-in procedure at the Smiths Falls/Montague Airport (SFMA) to optimize several aspects of the system, such as passenger wait times, profit maximization, and resource allocation efficiency.

Background:

We know that SFMA primarily serves Air Canada, offering regional commuter flights every hour to Ottawa and provincial flights to Calgary every six hours. The simulation considers various factors such as flight capacities, ticket costs, operational expenses, and check-in agent payments.

Arrival at the Airport:

Passenger arrivals follow Poisson processes for regional commuters and normal distribution for provincial flights. The number of bags each passenger carries can be determined by a geometric distribution. Geometric distribution models a series of independent events, meaning future outcomes are unaffected by past results.

Station 1: Check-in Counters:

The airline has six counters for check-in. Business-class and coach passengers have separate lines. Service times for printing boarding passes, checking bags, and handling other problems or delays are considered stochastic because these are random probability distributions.

Station 2: Security Screening:

Just like the check-in counter, there are separate lines for business class and coach passengers with different screening machines, 1 for business and 2 for coach. Screening time follows an exponential distribution as it is used to model the time between events in a process that occurs continuously and independently at a constant average rate.

Station 3: At the Gate:

Provincial Passengers:

- If a provincial passenger leaves the screening area before or at the plane's departure time, they are assumed to have made the flight.
- If a provincial passenger is not at the gate when the flight takes off, they are considered to have missed the flight and will leave the airport.
- If a provincial passenger arrives at least 90 minutes before the scheduled departure, the airline will refund the ticket price due to airport overcrowding; otherwise, no refund is provided.

Commuter Passengers:

- Commuter passengers cannot "miss a flight" since their tickets are not for a specific flight.
- The gate area serves as a third waiting queue for commuter passengers.
- Commuter passengers will wait patiently and board the next flight with an available spot.

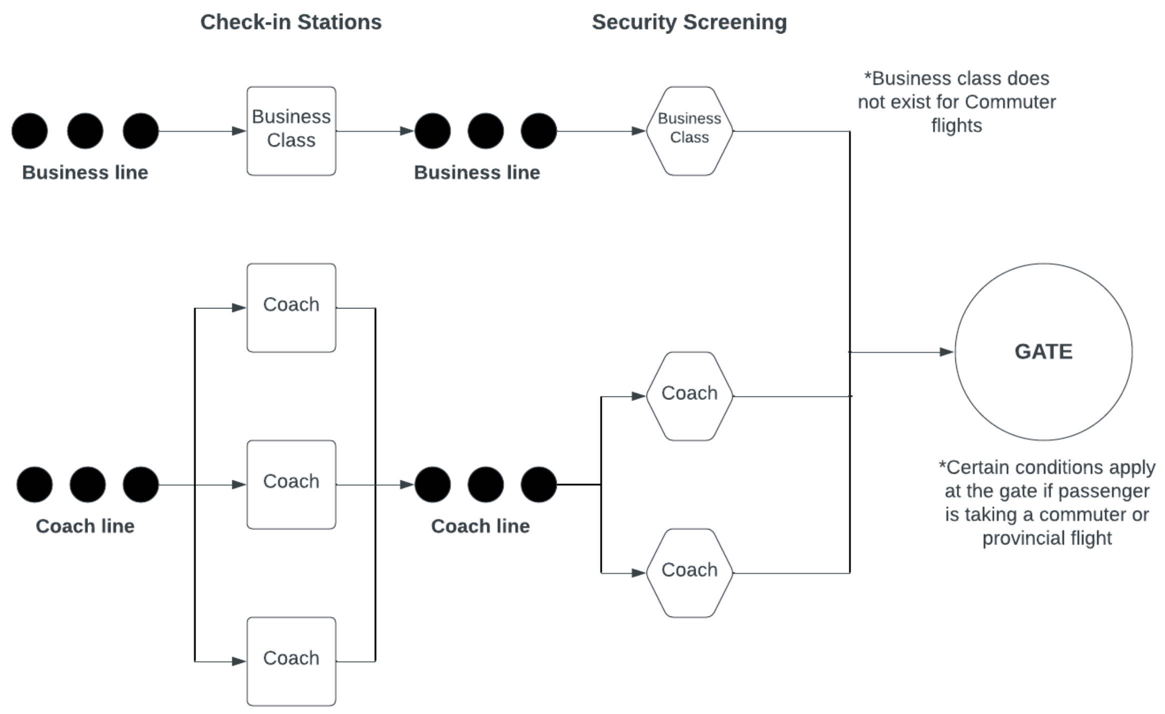

Check-in Stations
Security Screening
Business Class
Business line
Business line
Business Class
*Business class does not exist for Commuter flights
Coach
Coach
Coach
Coach line
Coach line
Coach
Coach
GATE
*Certain conditions apply at the gate if passenger is taking a commuter or provincial flight

Figure 1: Problem Formulation

2.0 PROJECT OBJECTIVES AND PLAN

The objectives of this simulation include managing many flight types, maximizing profit, minimizing passenger check-in wait time, improving provincial passenger likelihood to catch flights, optimizing check-in agent distribution, and minimizing agent idle time.

To guide the simulation of the SFMA check-in system, I formulated a plan:

1. Design and implement simulation classes for passenger arrival, check-in counters, security screening, and gate procedures for both coach and business passengers.

2. Run the needed experiments and gather data to assess the performance of the system.

3. Propose new optimization strategies to help enhance the system's performance.

4. Analyze system success rate and productivity based on returned profit, average check-in time, average screening time, and average waiting time for the simulation duration.

5. Make necessary system adjustments (e.g. increasing counters, reallocation check-in agents, adjusting flight times etc) and document the findings appropriately.

3.0 MODEL CONCEPTUALIZATION

1. Two types of flights (commuter and provincial) with different passenger capacities and reservation policies.
2. Arrival processes following Poisson and normal distributions respectively.
3. Check-in counter allocation, service times, and stochastic delays.
4. Check-in counties and security screening stations with dedicated queues for business-class and coach passengers.
5. Gate procedures with specific handling for provincial and commuter passengers.

4.0 MODEL TRANSLATION

4.1 Simulation Language Choice

Java is chosen for its versatility, object-oriented features, and extensive libraries. This includes components for random number generation, data structures, and concurrency support which allows the simulation to model parallel processes effectively, such as simultaneous check-ins, screenings, and gate procedures.

It also offers platform independence which ensures that the simulation can run seamlessly across various operating systems and supports the development of complex simulations. Java allows us to divide the main parts of the simulation into their respective classes to avoid high coupling and promote cohesion.

Lastly, I have the most personal experience using the Java language for projects and I am comfortable executing the simulation in an object-oriented fashion. The object-oriented nature of Java promotes code organization and readability, facilitating the modelling of real-world entities.

4.2 Model Implementation

1. Main.java -

The `Main` class serves as the heart of an airport check-in simulation, orchestrating the flow of passengers through various stages of airport processing, from check-in to security screening and finally to gate procedures. It simulates a realistic airport operation environment by incorporating key factors such as passenger arrival times, flight types, and service times. Here's a detailed overview of its functions:

- **Initialization:** Sets up the simulation environment, including the duration of the simulation and the number of check-in counters for both business and coach classes. This initial setup involves gathering input from the user to customize the simulation parameters.

- **Passenger Generation Loop:** For each minute of the simulation time, the program generates passengers based on predetermined probabilities. It determines whether they are commuter or provincial, their class (business or coach), and the number of bags they carry, reflecting a realistic mix of passenger types and behaviours.

- **Processing Through Simulation Components:**
    - Check-In Counter: Passengers are first processed through check-in counters where their arrival is logged, and boarding passes and bag checks are conducted.
    - Security Screening: Following check-in, passengers proceed to security screening, where they undergo a simulated screening process with service times generated to mirror real-life variability.
    - Gate Procedure: Finally, passengers are processed at the gate, where their eligibility to board the next available flight is determined based on their arrival time at the gate and flight departure schedules.

- **Revenue and Costs Calculation:** The simulation calculates total revenue by considering ticket prices for commuter and provincial flights and differentiating between business and coach classes. It also calculates total operational costs, including flight operation costs and check-in agent costs, based on the number of flights within the simulation period and the duration of operation.

- **Profit Calculation:** Determines the profit by subtracting total operational costs from total revenue, offering insights into the financial performance of the airport operations within the simulated environment.

- **Performance Metrics:** Calculates and displays average service times for check-in, security screening, and waiting times at the gate, providing metrics to evaluate the efficiency and effectiveness of the simulated airport processes.

- **User Interaction:** Engages the user by prompting for input parameters such as simulation time and the number of check-in counters, making the simulation interactive and customizable based on user preferences.

2. **Passenger.java -**

- Represents a passenger within a simulation environment, focusing on airport check-in processes. It holds key passenger attributes including:
    - **`id`:** A unique identifier for each passenger.
    - **`arrivalTime`:** The simulated time a passenger arrives at the airport.
    - **`checkInTime`**: The time at which the passenger checks in.
    - **`isCommuter`:** A boolean indicating if the passenger is commuting (as opposed to being a provincial traveller).
    - **`isBusinessClass`:** A boolean indicating if the passenger is travelling in business class.
    - **`numBags`:** The number of bags the passenger is carrying.
    - **`flightDepartureTime`**: The departure time of the passenger's flight.

- Utilizes a static **`passengerCount`** to ensure each `Passenger` instance receives a unique ID upon creation.
- The constructor initializes a passenger with the provided attributes and uses the `PassengerArrival` class to dynamically generate an accurate arrival time based on the commuter status.
- Provides getter methods for accessing passenger details such as ID, arrival and check-in times, commuter status, business class status, number of bags, and flight departure time.
- The class structure enables detailed simulation of passenger flow through airport check-in procedures, including how passengers of different types (commuter/provincial and business/coach) are processed and managed within the simulation.

3. PassengerArrival.java -

This Java class is central to simulating the dynamics of passenger arrivals at an airport within the context of a simulation, characterized by its ability to generate realistic variability in passenger arrival times and classify passengers based on specific criteria. The key functionalities and characteristics of this class are as follows:

- **Random Arrival Time Generation:** Utilizes both exponential and normal distribution models to simulate the arrival times of passengers. The exponential distribution is applied for commuter passengers, reflecting a more constant flow, while the normal distribution is used for provincial passengers, indicating a more varied arrival pattern.

- **Passenger Classification:**
    - Commuter vs. Provincial: Determines whether a passenger is a commuter or provincial, based on the current simulation time. The classification influences the distribution used to generate the passenger's arrival time.
    - Business Class Probability: Assigns a higher probability for provincial passengers to be travelling in business class compared to commuters, reflecting the different travel patterns and preferences between these groups.

- **Service Time Generation:**
    - Exponential Random Value: Generates service times for activities such as baggage check-in, represented by an exponential distribution to simulate variability in processing times.
    - Normal Random Value: Also capable of generating times based on a normal distribution for other aspects of the airport process, further adding to the realism of the simulation.

- **Passenger Bag Count:** Implements a geometric distribution to determine the number of bags a passenger carries, with success biases adjusted based on whether the passenger is commuter or provincial, thereby reflecting the differing likelihoods of baggage amounts between these two types of travellers.

- **Integration with Passenger Class:** The arrival time generated by this class is intended to be used in conjunction with the `Passenger` class constructor to set each passenger's arrival time based on their classification as commuter or provincial.

- **Utilization of Randomness:** Employs the `Random` class extensively to underpin the probabilistic models and distributions used, ensuring a dynamic and varied simulation output that mirrors the unpredictability of real-world airport operations.

4. **CheckInCounter.java -**

The CheckInCounter class is designed to simulate the check-in process at an airport, specifically catering to different passenger classes (business and coach) and managing the variability of service times through probabilistic models. Here's an overview of how this class functions:

- **Initialization:** Upon instantiation, the class initializes two queues: one for business class passengers and another for coach passengers. It also sets up a Random instance for generating random numbers and initializes the number of available counters for both business and coach classes based on constructor parameters.

- **Service Time Simulation:** It simulates service times for various check-in activities using an exponential distribution, which is a common probabilistic model for service processes. This includes time taken to print boarding passes, check bags, and handle other delays. The method `generateServiceTime(double avgServiceTime)` is responsible for generating these service times based on the average service time provided as an argument.

- **Process Passengers:** The `processPassenger(Passenger passenger)` method is the core of this class, where passengers are processed through the check-in counters. Each passenger is assigned to the appropriate queue based on their class. If a queue reaches its capacity (determined by the number of available counters), an overflow queue is created, simulating a real-world scenario where excess passengers wait for the next available counter.

- **Queue Management:** The class effectively manages two separate queues for business class and coach passengers, accommodating the check-in process according to the specific service counter allocations for each class.

- **Service Time Calculation and Accumulation:** As passengers are processed, the class calculates the total service time for each passenger, factoring in the time for printing boarding passes, checking bags, and additional delays. This total service time is then added to a cumulative total, and the passenger count is incremented to later calculate the average check-in time.

- **Average Check-In Time Calculation:** The `getAverageCheckInTime()` method calculates the average check-in time per passenger by dividing the total accumulated check-in time by the number of processed passengers. This metric provides insight into the efficiency of the check-in process over the simulation period.

- **Print Statements for Simulation Feedback:** Throughout the processing of passengers, the class outputs details to the console, such as queue status (e.g., when an overflow queue is created) and individual passenger check-in details (e.g., ID, class, number of bags, and check-in time). These print statements serve as a form of immediate feedback on the simulation's progression and outcomes.

5. **SecurityScreening.java -**

The SecurityScreening class simulates security screening operations at the SFMA airport. Key features include:

- Separate queues for business class and coach passengers (businessClassQueue and coachQueue).
- Utilizes a `Random` object for simulation randomness.

- The constructor initializes queues and the random object.
- Method `processPassenger(Passenger passenger)` to enqueue passengers based on coach or business class status.
- Method to display the results of the simulation (displayResults()).

The `SecurityScreening` class in Java is designed to simulate the security screening process at an airport, differentiating between business class and coach passengers based on the availability of screening machines. Its functionality include:

- **Initialization:** Upon creation, the class initializes two queues: one for business class passengers and another for coach passengers, reflecting the segregation of passengers based on ticket class during the security screening process. A `Random` object is also instantiated to facilitate the generation of random service times.

- **Queue Management:** It maintains separate queues (`businessClassQueue` and `coachQueue`) for managing passengers awaiting security screening. This setup allows for differentiated processing, catering to the distinct lanes often observed in airports for business class and coach passengers.

- **Machine Allocation:** The class is configured with a predefined number of screening machines for both business class (`numBusinessMachines`) and coach (`numCoachMachines`), simulating the physical setup of an airport's security checkpoint. The default configuration includes one machine for business class and two for coach.

- **Service Time Simulation:** It simulates the service time for each passenger using an exponential distribution, a common model for service processes. This is aimed at capturing the inherent variability in how long each passenger takes to go through security screening, with `averageScreeningTime` set to three minutes.

- **Processing Passengers:** The `processPassenger(Passenger passenger)` method assigns each passenger to the appropriate queue based on their class. It then simulates the security screening process for the number of available machines, dequeuing passengers and calculating their service times. This method embodies the core functionality of the class, integrating queue management with service time simulation.

- **Screening Time Generation:** Utilizes `generateScreeningTime()` to compute a random screening time for each passenger, employing the `generateExponentialServiceTime(double averageServiceTime)` method to apply the exponential distribution based on the specified average service time.

- **Average Screening Time:** The class tracks the total screening time and the count of passengers processed to calculate the average screening time per passenger, providing insight into the efficiency of the security screening operation.

6. **GateProcedure.java -**

   The `GateProcedure` class in Java is tasked with managing the final stage before boarding for passengers, aligning their processing with flight departure times, and distinguishing between commuter and provincial passengers based on the simulation's timing. Here's a detailed breakdown of its functionalities:

   - **Queue Management:** Maintains two separate queues for commuter and provincial passengers, allowing for differentiated handling based on passenger type.

   - **Flight Departure Time:** Stores a simulated flight departure time, which is crucial for determining whether passengers make their flight or need to wait for the next one.
   - **Average Waiting Time Calculation:** Calculates the average waiting time for all processed passengers, providing insights into the efficiency of the gate procedure and the passenger experience.

   - **Passenger Processing:**
     - For Provincial Passengers: Checks if a provincial passenger has arrived early enough to catch their flight. If not, it determines whether they missed the flight and potentially issues a refund based on the simulation's logic for early arrivals facing airport congestion.

    - For Commuter Passengers: Manages commuter passengers by having them wait for the next available flight if they arrive after their scheduled departure, simulating the continuous flow of commuter flights.

- **Passenger Queue Assignment:** Based on whether a passenger is a commuter or provincial, they are assigned to the respective queue. This method also calculates and displays each passenger's waiting time, updating the total waiting time and passenger count for later calculation of average waiting times.

- **Flight Time Calculation:** Implements a method to calculate the next available flight time for a passenger based on their type (commuter or provincial) and the current simulation time. This calculation accounts for commuter flights departing every half hour and provincial flights departing every six hours, ensuring passengers are allocated to flights realistically.

4.3 Design Diagrams

4.3.1 UML Class Diagram

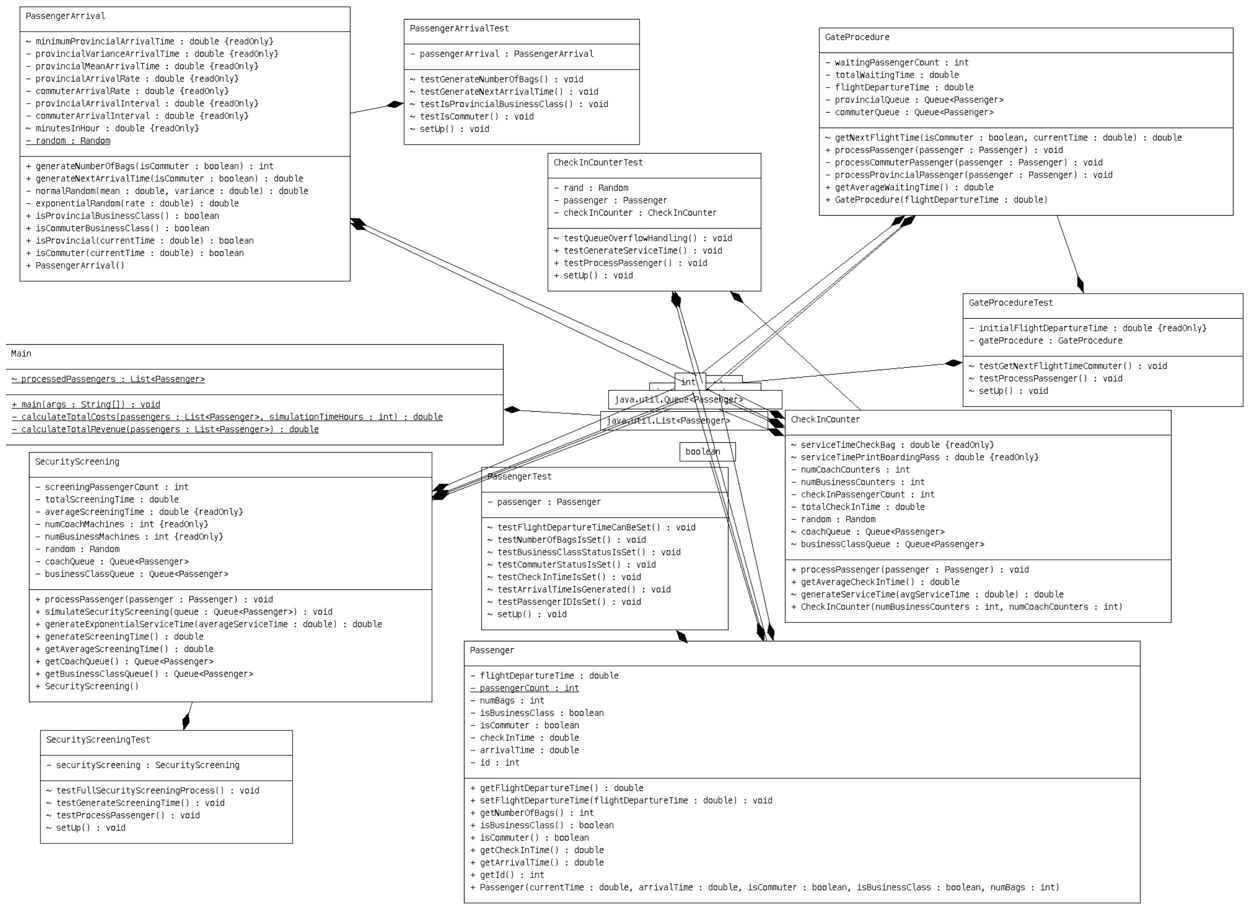

PassengerArrival
~ minimumProvincialArrivalTime : double {readOnly}
- provincialVarianceArrivalTime : double {readOnly}
- provincialMeanArrivalTime : double {readOnly}
- provincialArrivalRate : double {readOnly}
- commuterArrivalRate : double {readOnly}
- provincialArrivalInterval : double {readOnly}
- commuterArrivalInterval : double {readOnly}
~ minutesInHour : double {readOnly}
- random : Random
+ generateNumberOfBags(isCommuter : boolean) : int
+ generateNextArrivalTime(isCommuter : boolean) : double
- normalRandom(mean : double, variance : double) : double
- exponentialRandom(rate : double) : double
+ isProvincialBusinessClass() : boolean
+ isCommuterBusinessClass() : boolean
+ isProvincial(currentTime : double) : boolean
+ isCommuter(currentTime : double) : boolean
+ PassengerArrival()
PassengerArrivalTest
- passengerArrival : PassengerArrival
~ testGenerateNumberOfBags() : void
~ testGenerateNextArrivalTime() : void
~ testIsProvincialBusinessClass() : void
~ testIsCommuter() : void
~ setUp() : void
GateProcedure
- waitingPassengerCount : int
- totalWaitingTime : double
- flightDepartureTime : double
- provincialQueue : Queue<Passenger>
- commuterQueue : Queue<Passenger>
~ getNextFlightTime(isCommuter : boolean, currentTime : double) : double
+ processPassenger(passenger : Passenger) : void
- processCommuterPassenger(passenger : Passenger) : void
- processProvincialPassenger(passenger : Passenger) : void
+ getAverageWaitingTime() : double
+ GateProcedure(flightDepartureTime : double)
CheckInCounterTest
- rand : Random
- passenger : Passenger
- checkInCounter : CheckInCounter
~ testQueueOverflowHandling() : void
+ testGenerateServiceTime() : void
+ testProcessPassenger() : void
+ setUp() : void
GateProcedureTest
- initialFlightDepartureTime : double {readOnly}
- gateProcedure : GateProcedure
~ testGetNextFlightTimeCommuter() : void
~ testProcessPassenger() : void
~ setUp() : void
Main
~ processedPassengers : List<Passenger>
+ main(args : String[]) : void
- calculateTotalCosts(passengers : List<Passenger>, simulationTimeHours : int) : double
- calculateTotalRevenue(passengers : List<Passenger>) : double
int
java.util.Queue<Passenger>
java.util.List<Passenger>
boolean
CheckInCounter
~ serviceTimeCheckBag : double {readOnly}
~ serviceTimePrintBoardingPass : double {readOnly}
- numCoachCounters : int
- numBusinessCounters : int
- checkInPassengerCount : int
- totalCheckInTime : double
- random : Random
~ coachQueue : Queue<Passenger>
~ businessClassQueue : Queue<Passenger>
+ processPassenger(passenger : Passenger) : void
+ getAverageCheckInTime() : double
~ generateServiceTime(avgServiceTime : double) : double
+ CheckInCounter(numBusinessCounters : int, numCoachCounters : int)
SecurityScreening
- screeningPassengerCount : int
- totalScreeningTime : double
- averageScreeningTime : double {readOnly}
- numCoachMachines : int {readOnly}
- numBusinessMachines : int {readOnly}
- random : Random
- coachQueue : Queue<Passenger>
- businessClassQueue : Queue<Passenger>
+ processPassenger(passenger : Passenger) : void
+ simulateSecurityScreening(queue : Queue<Passenger>) : void
+ generateExponentialServiceTime(averageServiceTime : double) : double
+ generateScreeningTime() : double
+ getAverageScreeningTime() : double
+ getCoachQueue() : Queue<Passenger>
+ getBusinessClassQueue() : Queue<Passenger>
+ SecurityScreening()
PassengerTest
- passenger : Passenger
~ testFlightDepartureTimeCanBeSet() : void
~ testNumberOfBagsIsSet() : void
~ testBusinessClassStatusIsSet() : void
~ testCommuterStatusIsSet() : void
~ testCheckInTimeIsSet() : void
~ testArrivalTimeIsGenerated() : void
~ testPassengerIDIsSet() : void
~ setUp() : void
Passenger
- flightDepartureTime : double
- passengerCount : int
- numBags : int
- isBusinessClass : boolean
- isCommuter : boolean
- checkInTime : double
- arrivalTime : double
- id : int
+ getFlightDepartureTime() : double
+ setFlightDepartureTime(flightDepartureTime : double) : void
+ getNumberOfBags() : int
+ isBusinessClass() : boolean
+ isCommuter() : boolean
+ getCheckInTime() : double
+ getArrivalTime() : double
+ getId() : int
+ Passenger(currentTime : double, arrivalTime : double, isCommuter : boolean, isBusinessClass : boolean, numBags : int)
SecurityScreeningTest
- securityScreening : SecurityScreening
~ testFullSecurityScreeningProcess() : void
~ testGenerateScreeningTime() : void
~ testProcessPassenger() : void
~ setUp() : void

Figure 2: Class Diagram

4.3.2 UML Sequence Diagram

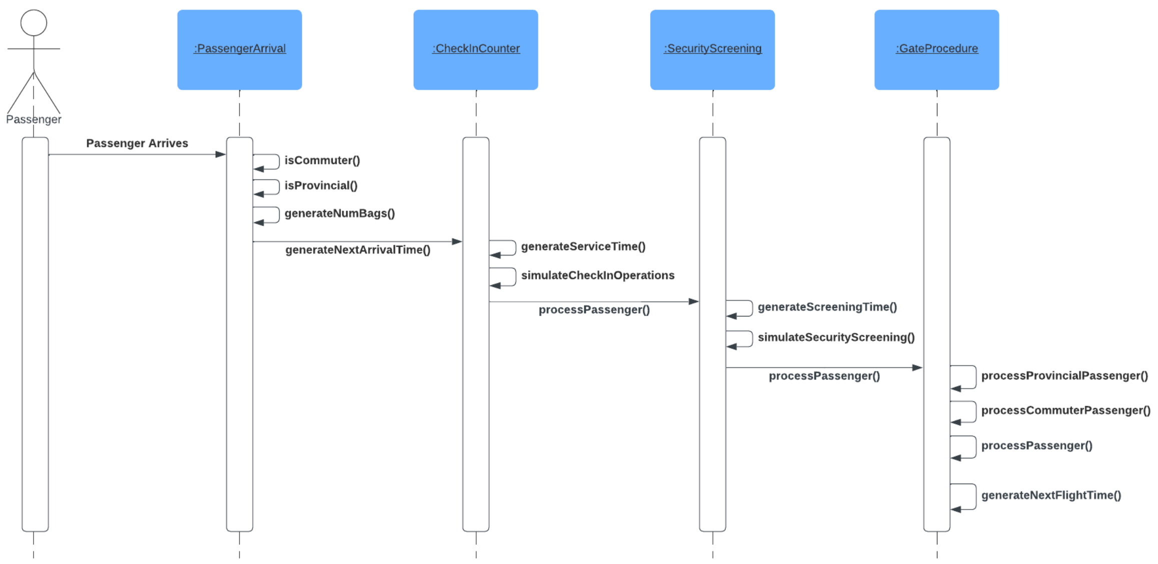


Figure 3: Sequence Diagram

4.0 DATA COLLECTION AND INPUT MODELLING

4.1 Data Collection

The SFMA simulation intricately models the journey of passengers from their arrival at the airport to the final boarding call, traversing through essential processes like check-in, security screening, and gate procedures. Central to this simulation is a carefully structured architecture composed of several Java classes, each dedicated to a specific segment of the airport's operational process, facilitating both data collection and input modelling within this immersive simulation framework.

At the heart of data collection lies the Passenger Class, a pivotal entity that encapsulates the quintessential attributes of passengers, including their identification, arrival time, status as commuter or provincial, travel class, and baggage count. This class is instrumental in capturing the individual nuances of each passenger's experience as they navigate through the simulation.

Complementing the Passenger Class is the PassengerArrival Class, which adeptly models the nuances of the arrival dynamics at the airport. Utilizing probabilistic models, it simulates the influx of passengers, meticulously distinguishing between commuter and provincial passengers while also determining their class of travel. This class is fundamental in setting the initial conditions for the simulation, generating passenger attributes that reflect realistic arrival rates and distributions.

The simulation delves deeper into the airport's operational mechanics through the CheckInCounter, SecurityScreening, and GateProcedure Classes. Each class is meticulously designed to model a specific stage of the airport process, collecting data on the interaction between passengers and these stages, informed by the attributes generated by the PassengerArrival Class. These interactions include vital operational metrics such as service and waiting times, which are crucial for understanding the flow and constraints within the simulated airport environment.

4.2 Input Modelling

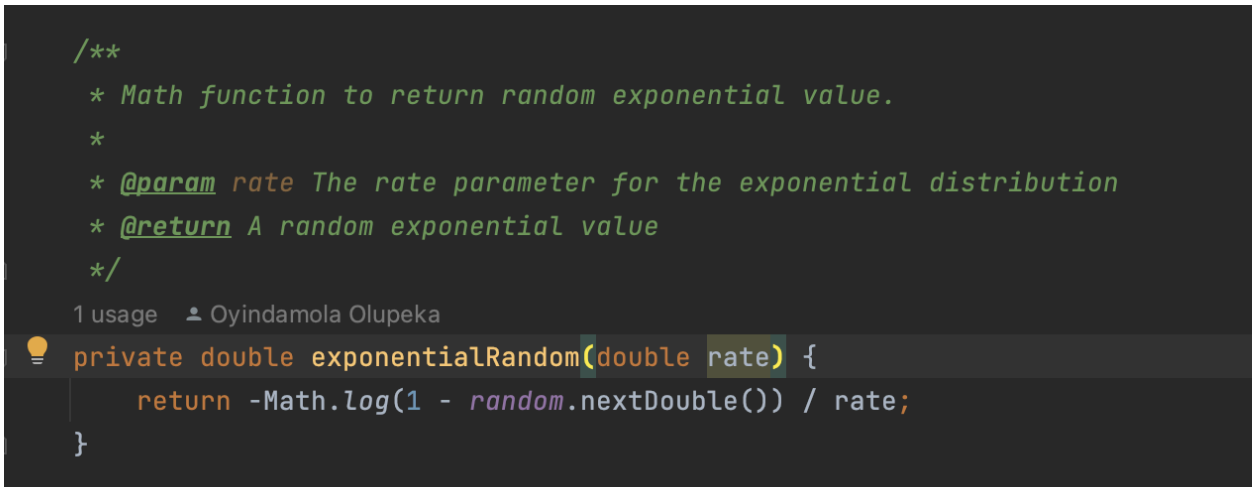
The generation of input in the SFMA simulator relies primarily on the PassengerArrival class, supplemented by the logic implemented in the Main class to simulate the passage of time and process passengers. Here's a detailed look at this process:

Time Simulation: The Main class simulates the passage of time within the simulation period, treating each iteration of its main loop as a minute. This continuous time model allows for the dynamic generation of passengers and their subsequent processing.

Passenger Generation: Within each time iteration, the Main class invokes the PassengerArrival class to determine if a new passenger arrives at that minute. The decision is based on probabilistic models:

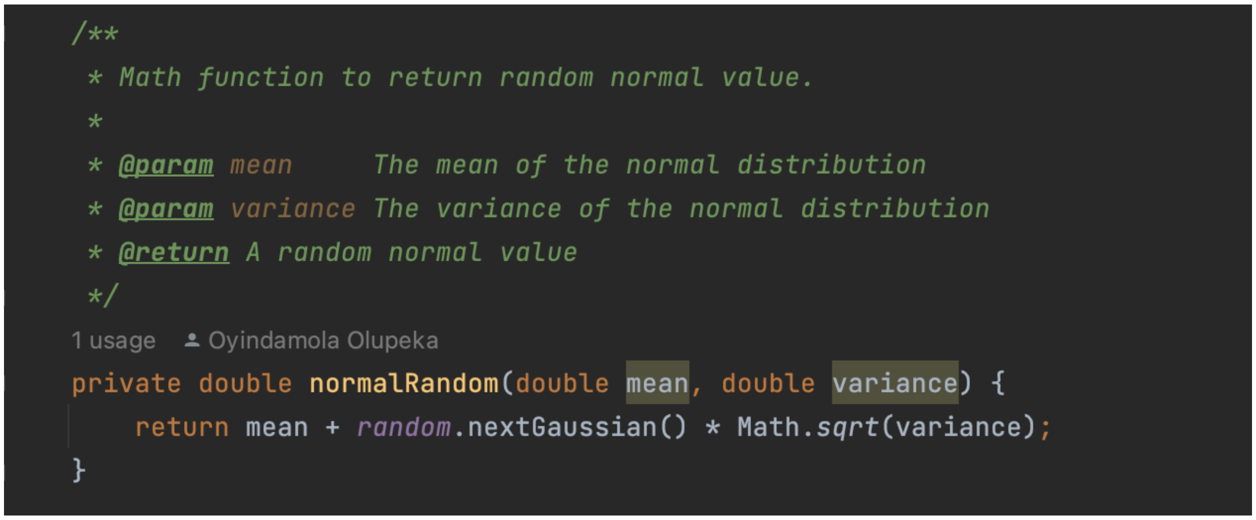
- Commuter vs. Provincial: The class randomly decides whether each new passenger is a commuter or provincial, simulating the variability of passenger types arriving at the airport.

- Arrival Times: For those identified as arriving, their arrival times are generated based on exponential distribution for commuters (to model the frequent but unpredictable arrival pattern) and normal distribution for provincials (to capture the less frequent but more variably timed arrivals).



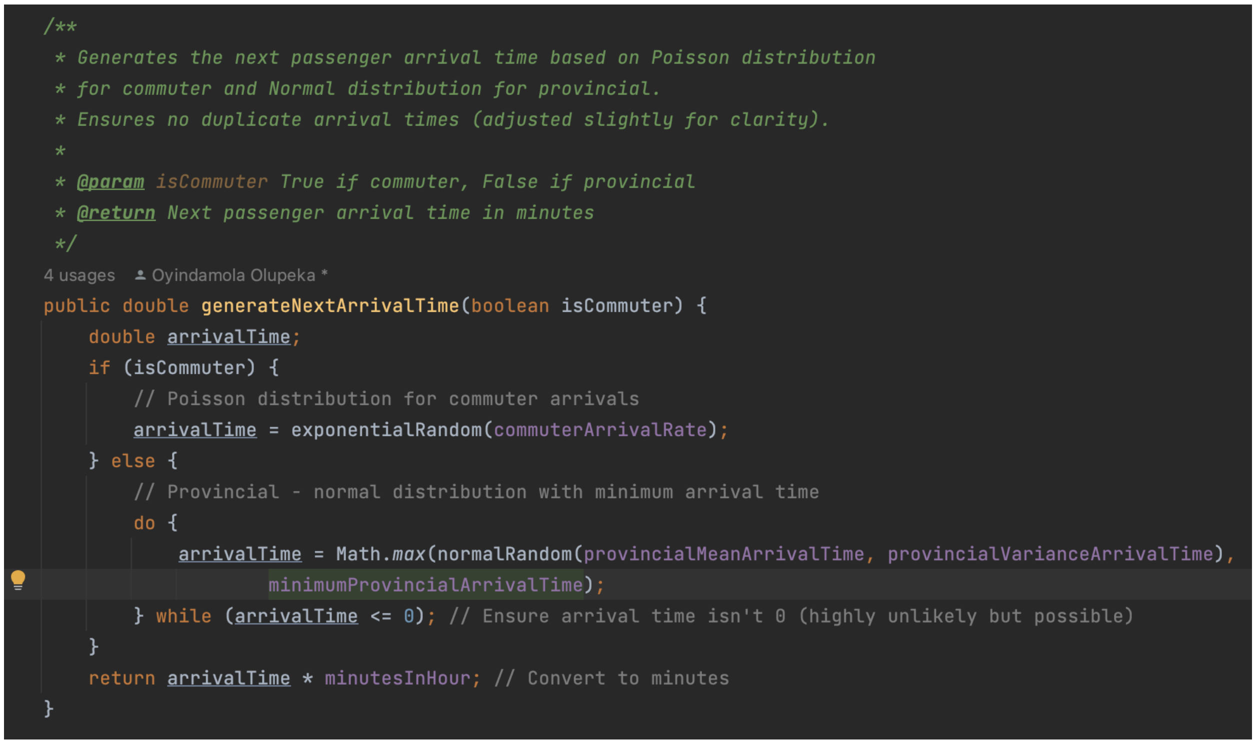
```java
/**
 * Math function to return random exponential value.
 *
 * @param rate The rate parameter for the exponential distribution
 * @return A random exponential value
 */
1 usage   Oyindamola Olupeka
private double exponentialRandom(double rate) {
    return -Math.log(1 - random.nextDouble()) / rate;
}
```

Figure 4: Exponential Random Code Excerpt



```java
/**
 * Math function to return random normal value.
 *
 * @param mean      The mean of the normal distribution
 * @param variance The variance of the normal distribution
 * @return A random normal value
 */
1 usage   Oyindamola Olupeka
private double normalRandom(double mean, double variance) {
    return mean + random.nextGaussian() * Math.sqrt(variance);
}
```

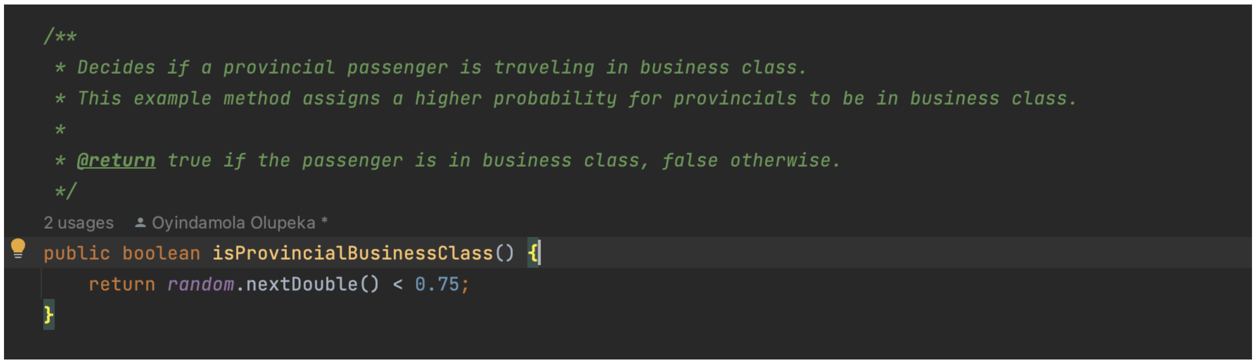
Figure 5: Normal Random Code Excerpt



```java
/**
 * Generates the next passenger arrival time based on Poisson distribution
 * for commuter and Normal distribution for provincial.
 * Ensures no duplicate arrival times (adjusted slightly for clarity).
 *
 * @param isCommuter True if commuter, False if provincial
 * @return Next passenger arrival time in minutes
 */
4 usages   Oyindamola Olupeka *
public double generateNextArrivalTime(boolean isCommuter) {
    double arrivalTime;
    if (isCommuter) {
        // Poisson distribution for commuter arrivals
        arrivalTime = exponentialRandom(commuterArrivalRate);
    } else {
        // Provincial - normal distribution with minimum arrival time
        do {
            arrivalTime = Math.max(normalRandom(provincialMeanArrivalTime, provincialVarianceArrivalTime),
                    minimumProvincialArrivalTime);
        } while (arrivalTime <= 0); // Ensure arrival time isn't 0 (highly unlikely but possible)
    }
    return arrivalTime * minutesInHour; // Convert to minutes
}
```

Figure 6: Arrival Time Generation Code Excerpt

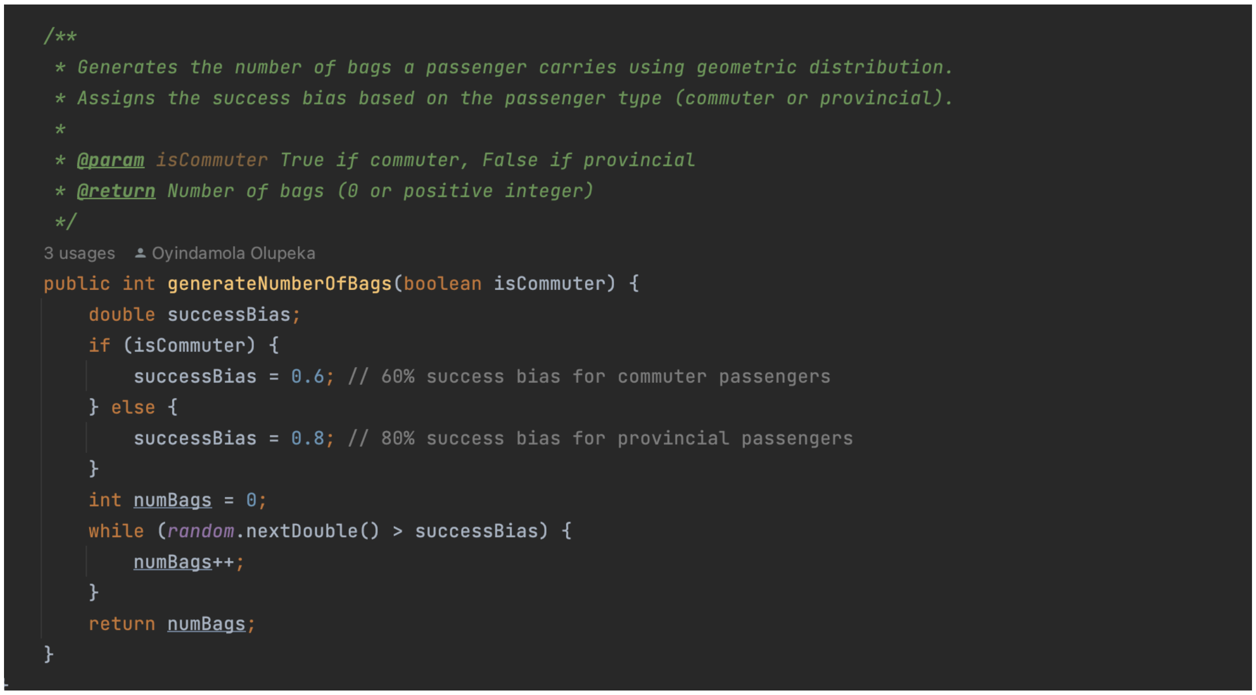
- Business Class for Provincial Passengers: It also probabilistically determines if a provincial passenger travels in business class, reflecting the higher likelihood of business travellers among provincial passengers. Set to a maximum rate of 75% for the experiment, therefore a maximum rate of 25% for coach class passengers.



```java
/**
 * Decides if a provincial passenger is traveling in business class.
 * This example method assigns a higher probability for provincials to be in business class.
 *
 * @return true if the passenger is in business class, false otherwise.
 */
2 usages   Oyindamola Olupeka *
public boolean isProvincialBusinessClass() {
    return random.nextDouble() < 0.75;
}
```

Figure 7: Provincial Business Class Probabilty Generation Code Excerpt

Baggage Generation: The number of bags each passenger carries is also generated based on a geometric distribution, varying by passenger type to add another layer of realism to the simulation.



```java
/**
 * Generates the number of bags a passenger carries using geometric distribution.
 * Assigns the success bias based on the passenger type (commuter or provincial).
 *
 * @param isCommuter True if commuter, False if provincial
 * @return Number of bags (0 or positive integer)
 */
3 usages   Oyindamola Olupeka
public int generateNumberOfBags(boolean isCommuter) {
    double successBias;
    if (isCommuter) {
        successBias = 0.6; // 60% success bias for commuter passengers
    } else {
        successBias = 0.8; // 80% success bias for provincial passengers
    }
    int numBags = 0;
    while (random.nextDouble() > successBias) {
        numBags++;
    }
    return numBags;
}
```

Figure 8: Passenger Bags Probabilty Generation Code Excerpt

Ensuring Unique Arrival Times: The simulator includes logic to ensure that each passenger has a unique arrival time, even if the difference is as minimal as a second, to avoid processing conflicts and accurately simulate an airport's operations. The arrival times are stored in instances of the Passenger class.

This detailed input generation mechanism, relying on a mix of deterministic scheduling and probabilistic models, allows the SFMA simulator to realistically model the flow of passengers through an airport, capturing the inherent variability and complexity of such systems.

5.0 MODEL VERIFICATION AND VALIDATION

5.1 Verification

Verification of the SFMA simulation involves ensuring that the implemented model correctly represents the conceptual model and operates as intended. The verification process for this simulation was approached using two methods:

1. **Code Review and Debugging:** Methodically reviewing the Java classes to check for logical errors, adherence to specifications, and proper implementation of the simulation processes (e.g., passenger arrival, check-in, security screening, and gate procedures). Debugging sessions help in identifying and fixing any anomalies or bugs in the simulation logic.

2. **Unit Testing:** Developing and executing unit tests for individual components, such as the Passenger, PassengerArrival, CheckInCounter, SecurityScreening, and GateProcedure classes. This helps in verifying the functionality of each module independently, ensuring that they perform as expected under various scenarios. JUnit 5 is a widely used testing framework for Java applications, offering comprehensive support for writing and executing unit tests. It provides a variety of features and annotations that simplify the process of writing tests and validating the functionality of individual components within an application. By using JUnit 5 in combination with Java, you can ensure the reliability and correctness of your code by systematically testing individual components in isolation, identifying and addressing issues early in the development process. The JUnit test classes are as follows:

- PassengerTest.java - This class contains a series of unit tests for the `Passenger` class. For each test case, the `setUp` method is used to initialize a `Passenger` instance with predefined values for testing purposes. Then, assertions are made using JUnit's assertion methods (`assertTrue`, `assertEquals`) to verify the expected behavior of the `Passenger` class under different scenarios.

- PassengerArrivalTest.java - This class contains unit tests for the `PassengerArrival` class. For each test case, the `setUp` method initializes a `PassengerArrival` instance before each test, ensuring a clean state for testing. Then, assertions are made using JUnit's assertion methods (`assertTrue`) to verify the expected behavior of the `PassengerArrival` class under different scenarios.

- CheckInCounterTest.java - This class contains unit tests for the `CheckInCounter` class. For each test case, the `setUp` method initializes a `CheckInCounter` instance, a `Passenger` instance, and a `Random` instance before each test. The `CheckInCounter` instance is initialized with a specific configuration (e.g., number of business and coach counters), and the `Passenger` instance represents a sample passenger with predefined attributes. Assertions are made using JUnit's assertion methods (`assertTrue`, `assertEquals`) to validate the behavior of the `CheckInCounter` class under various scenarios.

- SecurityScreening.java - This class contains unit tests for the `SecurityScreening` class. For each test case, the `setUp` method initializes a `SecurityScreening` instance before each test. The `SecurityScreening` instance represents the object being tested. Assertions are made using JUnit's assertion methods (`assertEquals`, `assertTrue`) to validate the behavior of the `SecurityScreening` class under various scenarios.

- GateProcedure.java - This class contains unit tests for the `GateProcedure` class. In each test case, the `setUp` method initializes a `GateProcedure` instance before each test. The `GateProcedure` instance represents the object being tested. Assertions are made using JUnit's assertion methods (`assertEquals`) to validate the behavior of the `GateProcedure` class under various scenarios.

5.2 Validation

Validation, on the other hand, ensures that the simulation accurately reflects the real-world operations it aims to replicate.

For a simulation of 60 minutes (1 hour) with 2 available business class counters and 4 coach class counter for the entire duration. Validation was approached through:

Check-In Time

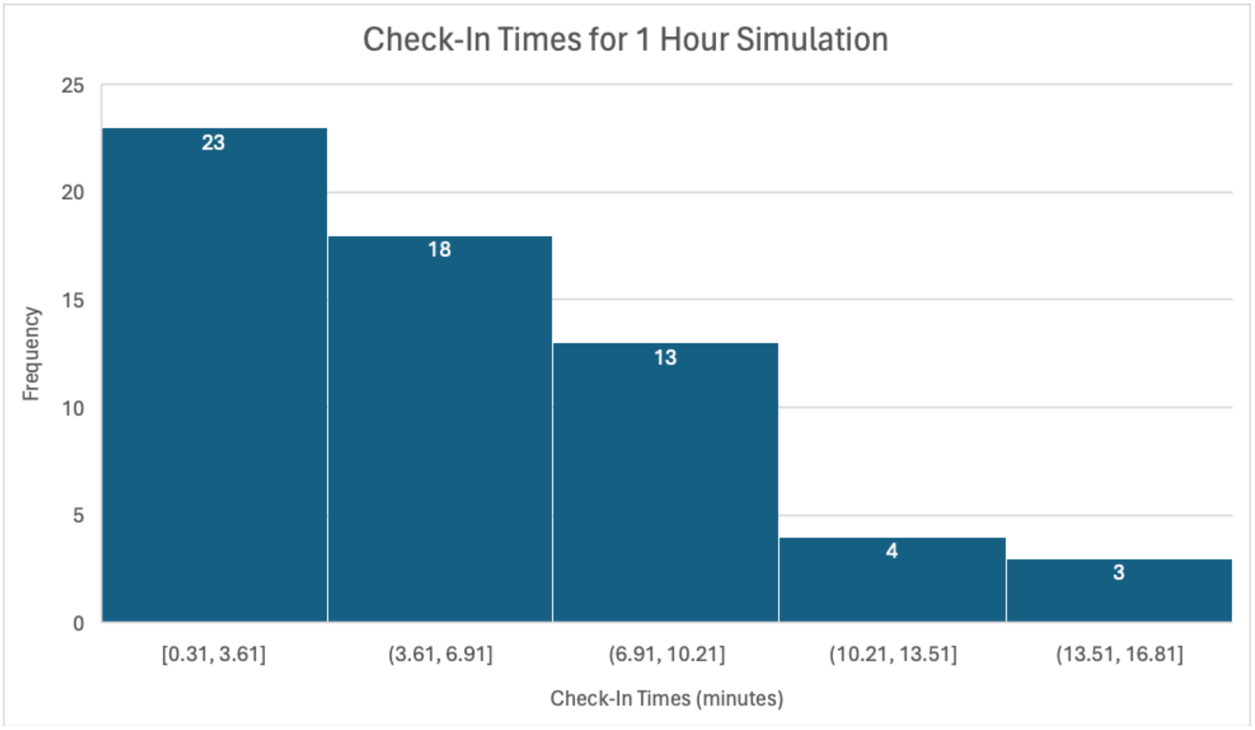


Figure 9: Check-In Times Histogram

The data represents check-in times and is a continuous variable. Based on the shape of this plot, the sample data fits an exponential distribution.

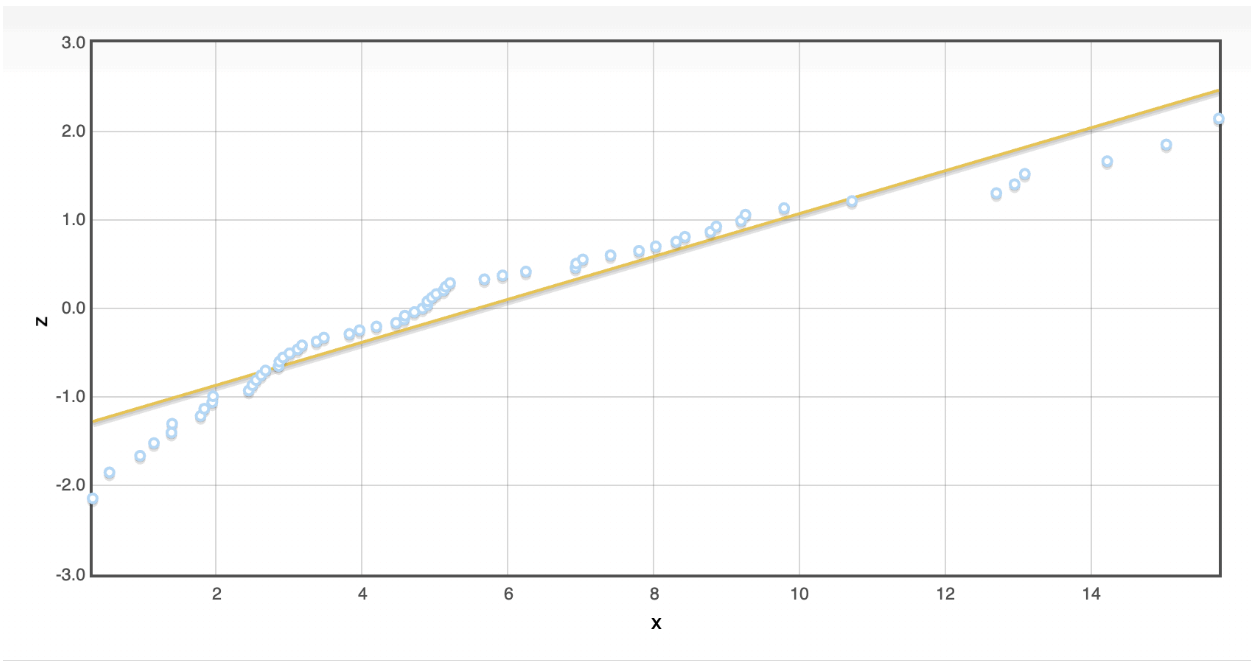


Figure 10: Check-In Times Q-Q Plot

The linearity of this plot further confirms that an exponential distribution for the check-in times is a good fit, and the parameter estimation is accurate since the slope of the plot is almost exactly one.

Security Screening Time

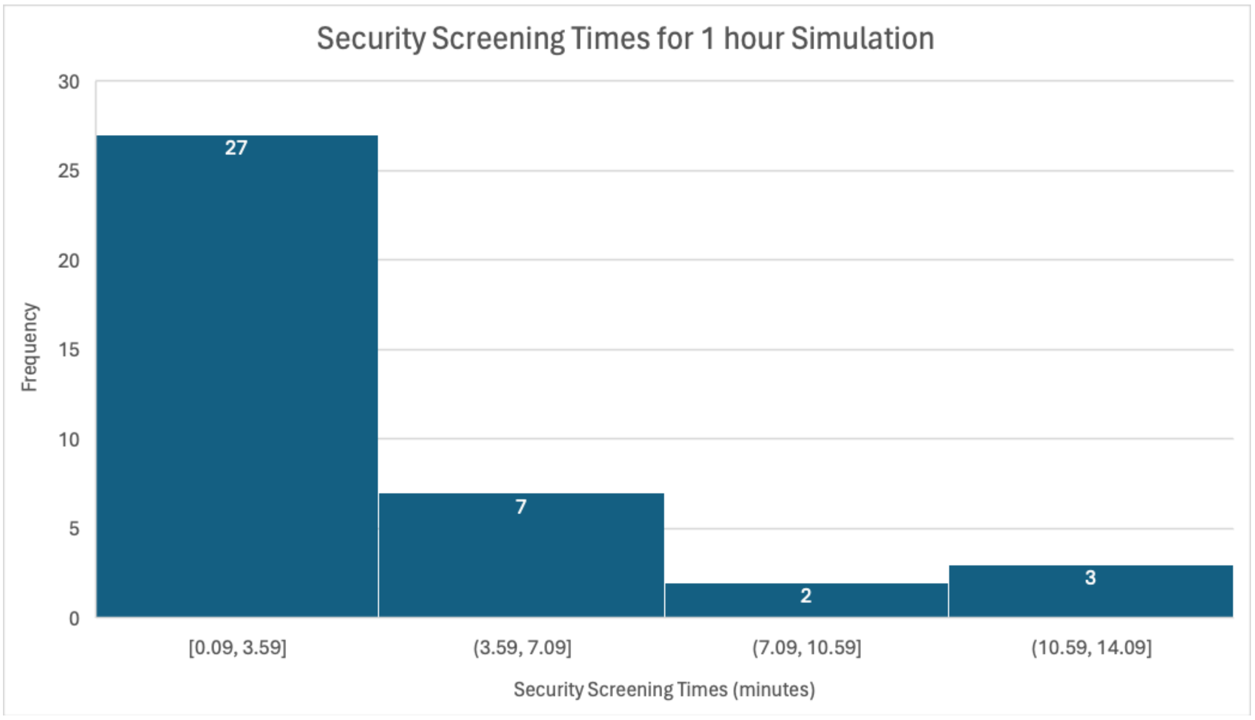


Figure 11: Security Screening Time Histogram

The data represents screening times and is a continuous variable. Based on the shape of this plot, the sample data does seem to fit an exponential distribution.

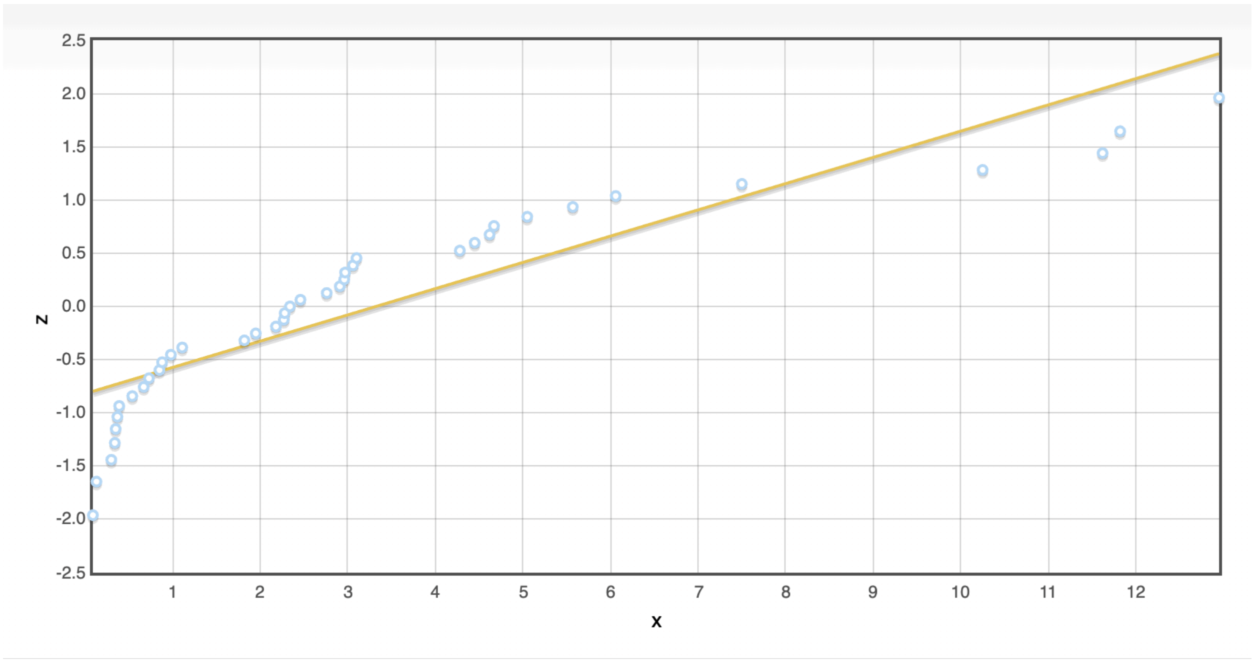


Figure 12: Security Screening Time Q-Q Plot

The linearity of this plot further confirms that an exponential distribution for the security screening times is a good fit, and the parameter estimation is accurate since the slope of the plot is almost exactly one.

Waiting Time

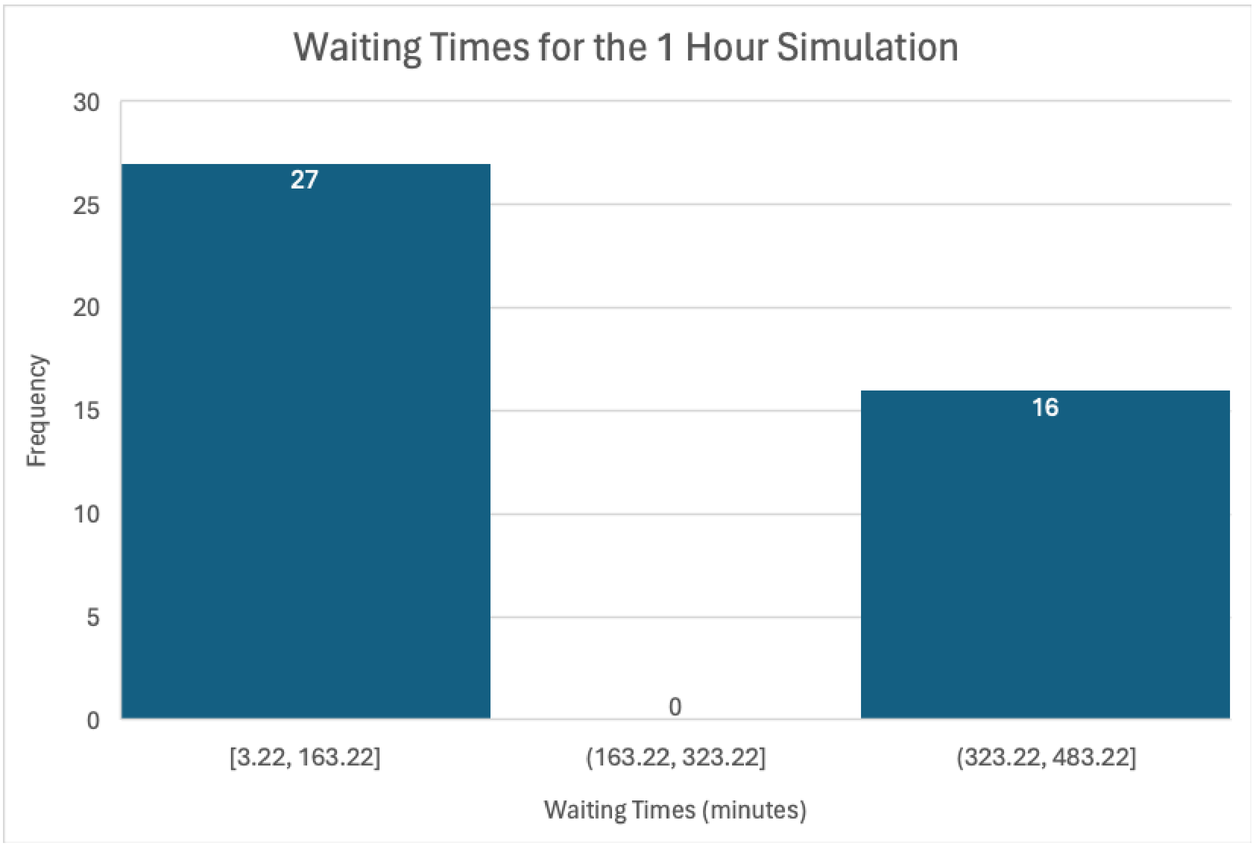


Figure 13: Waiting Time Histogram

The data represents waiting times of the simulation. Based on the shape of this plot, teh results are very split based on the set flight times (by the hour or on the half hour). More people are waiting less based on this simulation.

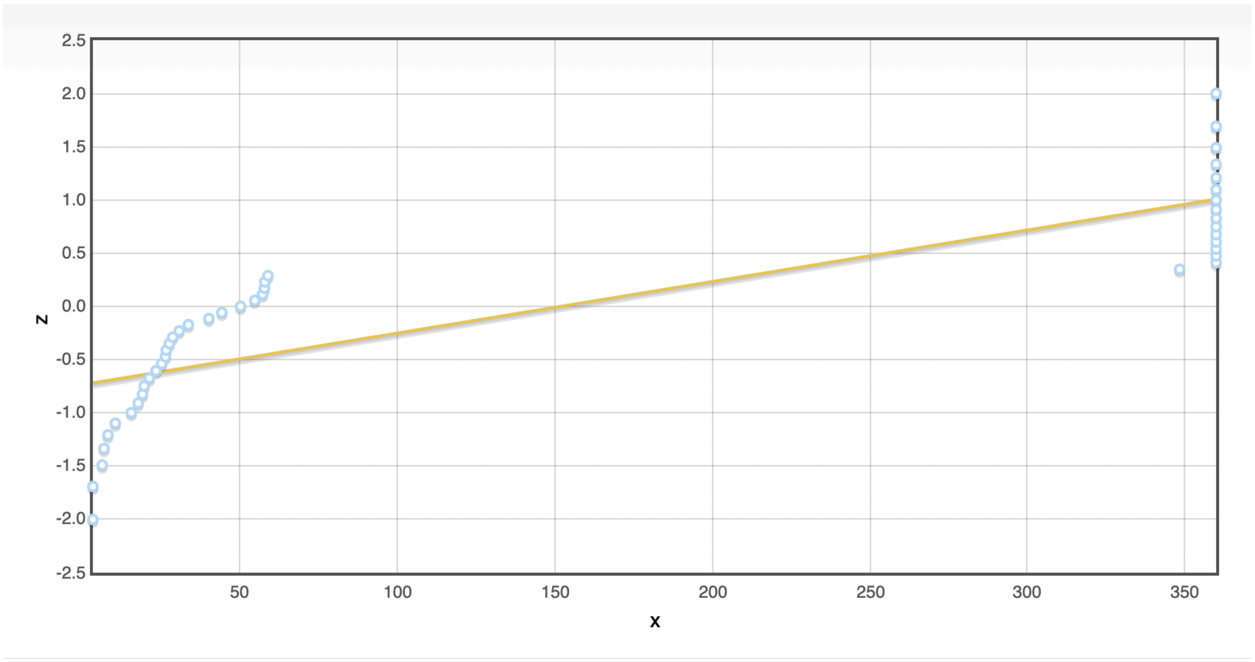


Figure 14: Waiting Time Q-Q Plot

The linearity of this plot further confirms that an exponential distribution for the check-in times is a good fit, and the parameter estimation is accurate since the slope of the plot is almost exactly one.

6.0 PRODUCTION RUNS AND ANALYSIS

To extract meaningful quantities of interest from the simulation, such as average waiting times, passenger throughput, and profitability metrics, conducting multiple independent replications of the simulation was crucial. This approach helped in assessing the variability of outcomes and ensured that the results were statistically reliable.

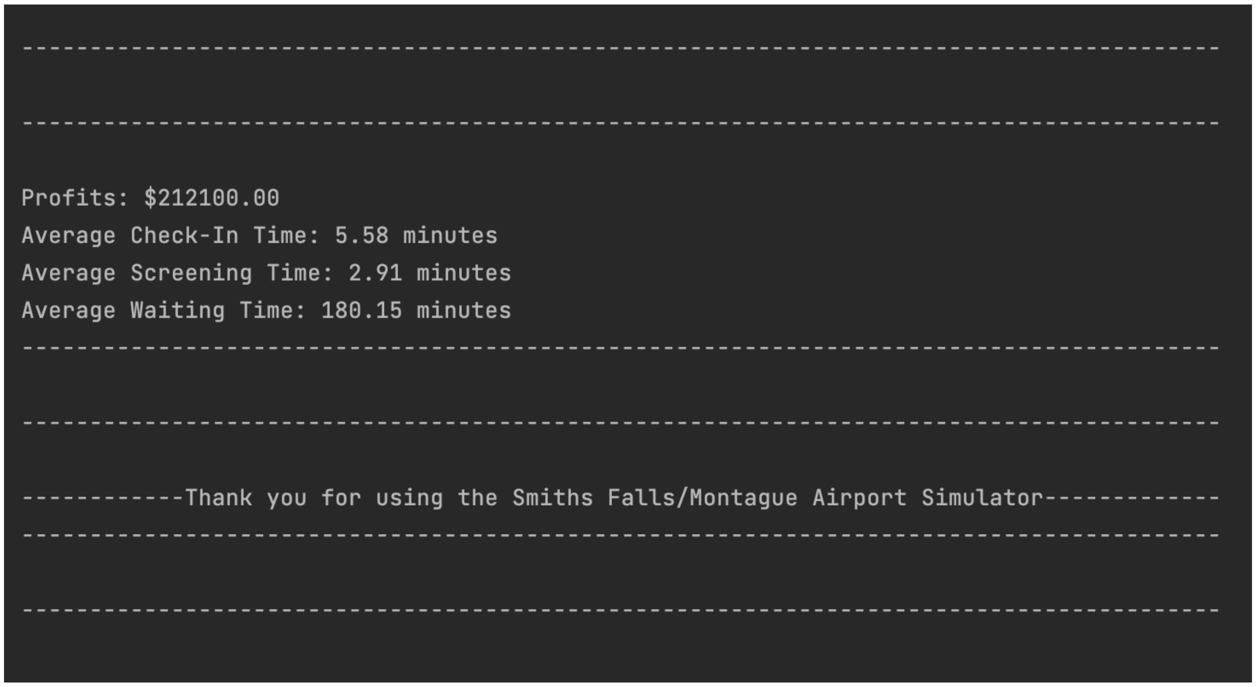
Independent Replications: Running the simulation multiple times with different random seed values ensured that each replication was independent. This variability allowed for the calculation of confidence intervals for the metrics of interest, providing insights into their stability and the simulation's overall reliability.

Initialization Phase: Considering an initialization phase was essential to allow the simulation to reach a steady state before collecting data for analysis. The length of this phase was determined by monitoring when key metrics stabilized. For the SFMA simulation, this involved observing when the passenger queues and processing times reached equilibrium.

Analysis: After conducting the replications and accounting for the initialization phase, the collected data were analyzed to extract average values and confidence intervals for the quantities of interest. This analysis provided insights into the effectiveness of airport operations and identified potential areas for improvement.

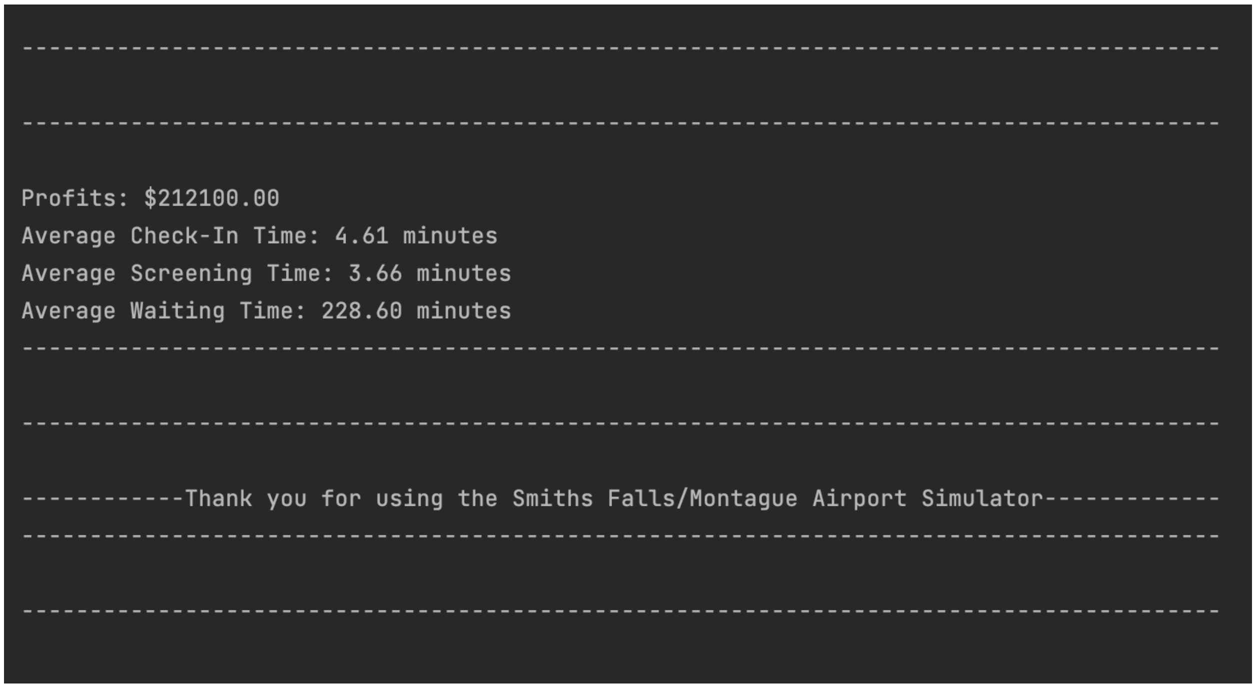
A number of replications were done to analyse the data. For simplification 5 different simulations are displayed based of different metric of available staff the the business class and coach counters but an identical simulation time of 1 Hour (60 minutes). The results are as follows:

Simulation 1: 1 Hour Simulation with 2 available Business class counters and 4 coach class counters



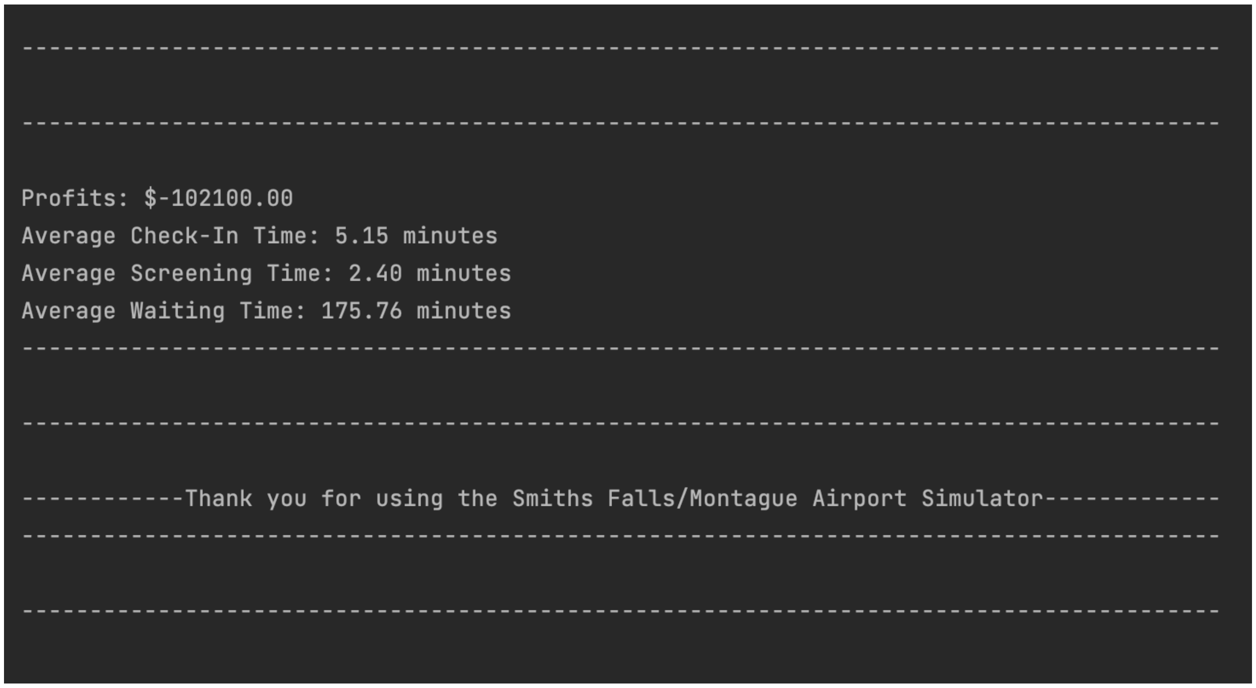
```
--------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------

Profits: $212100.00
Average Check-In Time: 5.58 minutes
Average Screening Time: 2.91 minutes
Average Waiting Time: 180.15 minutes
--------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------

------------Thank you for using the Smiths Falls/Montague Airport Simulator-------------
--------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------
```

Simulation 2: 1 Hour Simulation with 1 available Business class counters and 3 coach class counters



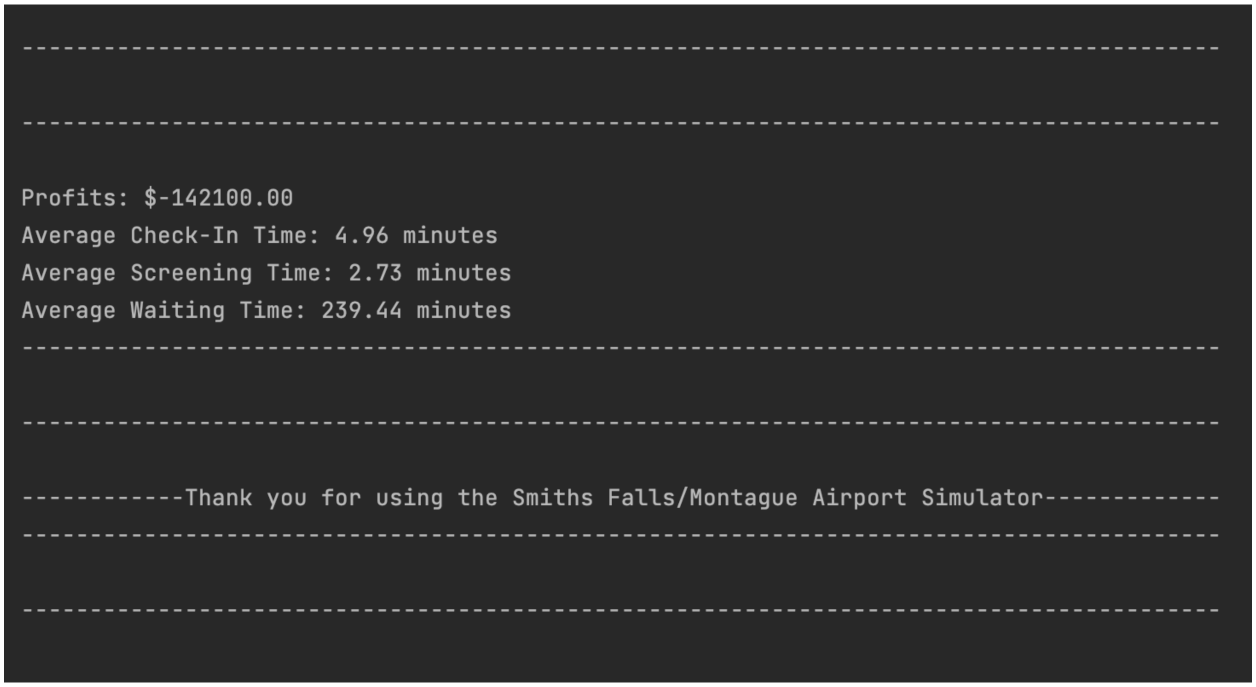
```
--------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------

Profits: $212100.00
Average Check-In Time: 4.61 minutes
Average Screening Time: 3.66 minutes
Average Waiting Time: 228.60 minutes
--------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------

------------Thank you for using the Smiths Falls/Montague Airport Simulator-------------
--------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------
```

Simulation 3: 1 Hour Simulation with 1 available Business class counters and 1 coach class counters



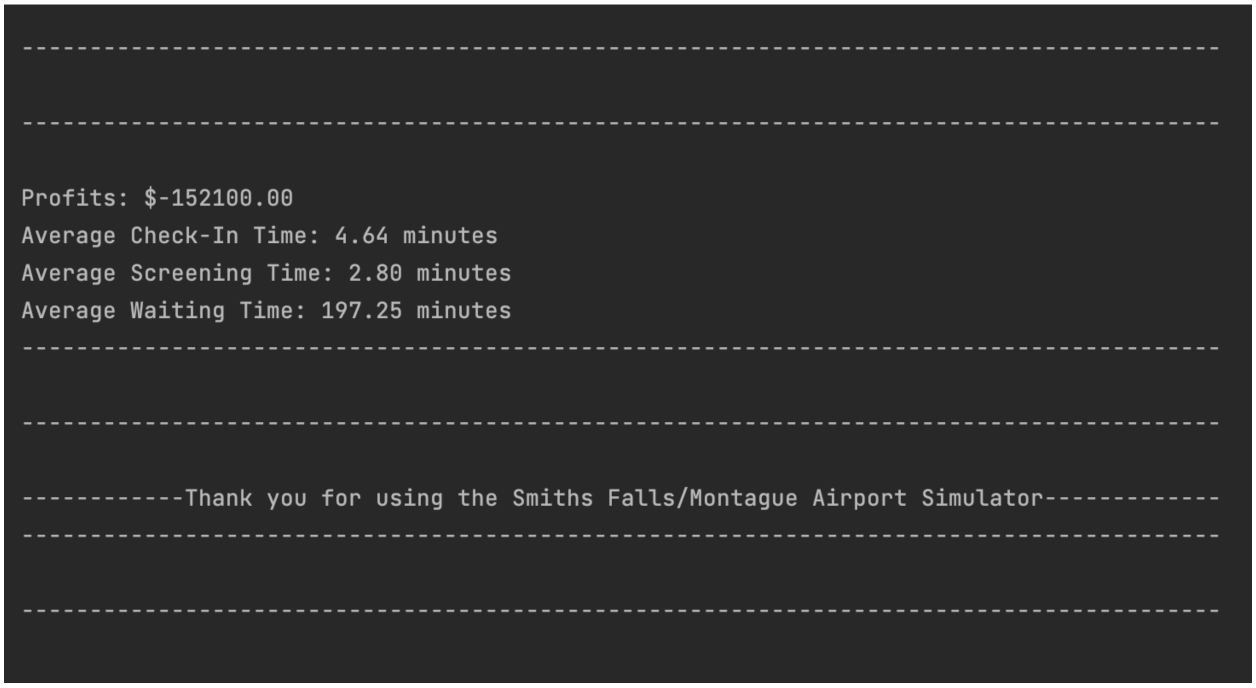
```
-----------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------

Profits: $-102100.00
Average Check-In Time: 5.15 minutes
Average Screening Time: 2.40 minutes
Average Waiting Time: 175.76 minutes
-----------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------

------------Thank you for using the Smiths Falls/Montague Airport Simulator-------------
-----------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------
```

Simulation 4: 1 Hour Simulation with 2 available Business class counters and 1 coach class counters



```
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------

Profits: $-142100.00
Average Check-In Time: 4.96 minutes
Average Screening Time: 2.73 minutes
Average Waiting Time: 239.44 minutes
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------

------------Thank you for using the Smiths Falls/Montague Airport Simulator-------------
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
```

Simulation 5: 1 Hour Simulation with 2 available Business class counters and 3 coach class counters



```
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------

Profits: $-152100.00
Average Check-In Time: 4.64 minutes
Average Screening Time: 2.80 minutes
Average Waiting Time: 197.25 minutes
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------

------------Thank you for using the Smiths Falls/Montague Airport Simulator-------------
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
```

Based on the results above, the number of counters available for the business class and coach class affects the profit, average check-in, screening, and waiting times. When maximising the available counters (as in Simulation 1) the SFMA makes the most profit and has the one of the lowest average waiting time. When maximising the available counters (as in Simulation 3) the SFMA makes the most no profit (a loss) and has a larger average waiting time.

7.0 ALTERNATIVE OPERATING POLICY

High-Level Implementation

The alternative design focuses on optimizing the passenger flow and reducing waiting times across the airport's operational processes, specifically targeting the check-in counters, security screening, and gate procedures. This is achieved by dynamically adjusting service resources based on real-time demand and introducing technological enhancements. For example, increasing the number of staff available for the check-in and security screening counters during peak times in the airport to aid productivity, reduce waiting times, and increase profits.

Detail-Level Implementation

1. Check-In Counters: Implement a dynamic allocation system for check-in counters based on real-time passenger flow data. During peak times, more counters are opened for service, especially for coach passengers who typically experience longer queues. Incorporate self-service kiosks for passengers with no checked baggage, further reducing demand on manned counters.
2. Security Screening: Increase the number of screening machines based on anticipated passenger volumes, with the flexibility to open additional lanes during peak periods.

Introduce advanced screening technology to expedite the process, such as more efficient scanners that reduce the need for physical checks.

3. Gate Procedures: Implement a staggered boarding process, prioritizing passengers by seat row numbers to streamline boarding and reduce gate crowding. Offer mobile updates to passengers about their boarding status and any changes to gate information or flight times.

This alternative policy aims to enhance system performance by reducing constraints, thereby improving passenger satisfaction and operational efficiency. Dynamic resource allocation ensures that services are scaled according to demand, minimizing idle resources during off-peak times and reducing waiting times during peak periods. Technological enhancements in security screening and the use of mobile updates for boarding contribute to a smoother passenger experience, potentially increasing airport throughput and profitability.

To evaluate the alternatives, comparisons key performance indicators (KPIs) such as average waiting times at check-in counters, security screening, and gates before and after implementing the changes. Additionally, passenger satisfaction surveys and operational cost analyses (considering both fixed and variable costs) provide insights into the effectiveness and economic viability of the recommended policy.

This proposed alternative operating policy introduces a more adaptable and efficient approach to managing passenger flow and resources in the airport simulation. By focusing on dynamic resource allocation and technological enhancements, the policy aims to reduce waiting times, improve passenger satisfaction, and increase operational efficiency. The evaluation through KPIs and cost-benefit analyses will further justify the implementation of these changes, offering a pathway for continuous improvement in airport operations.

8.0 CONCLUSION

In conclusion, the comprehensive overview provided in this final report encapsulates the characteristics of the simulation project, spanning its design, implementation, and analysis. Beginning with the simulation model and its components, a robust framework for evaluating operational performance within the context of passenger processing at an airport has been established.

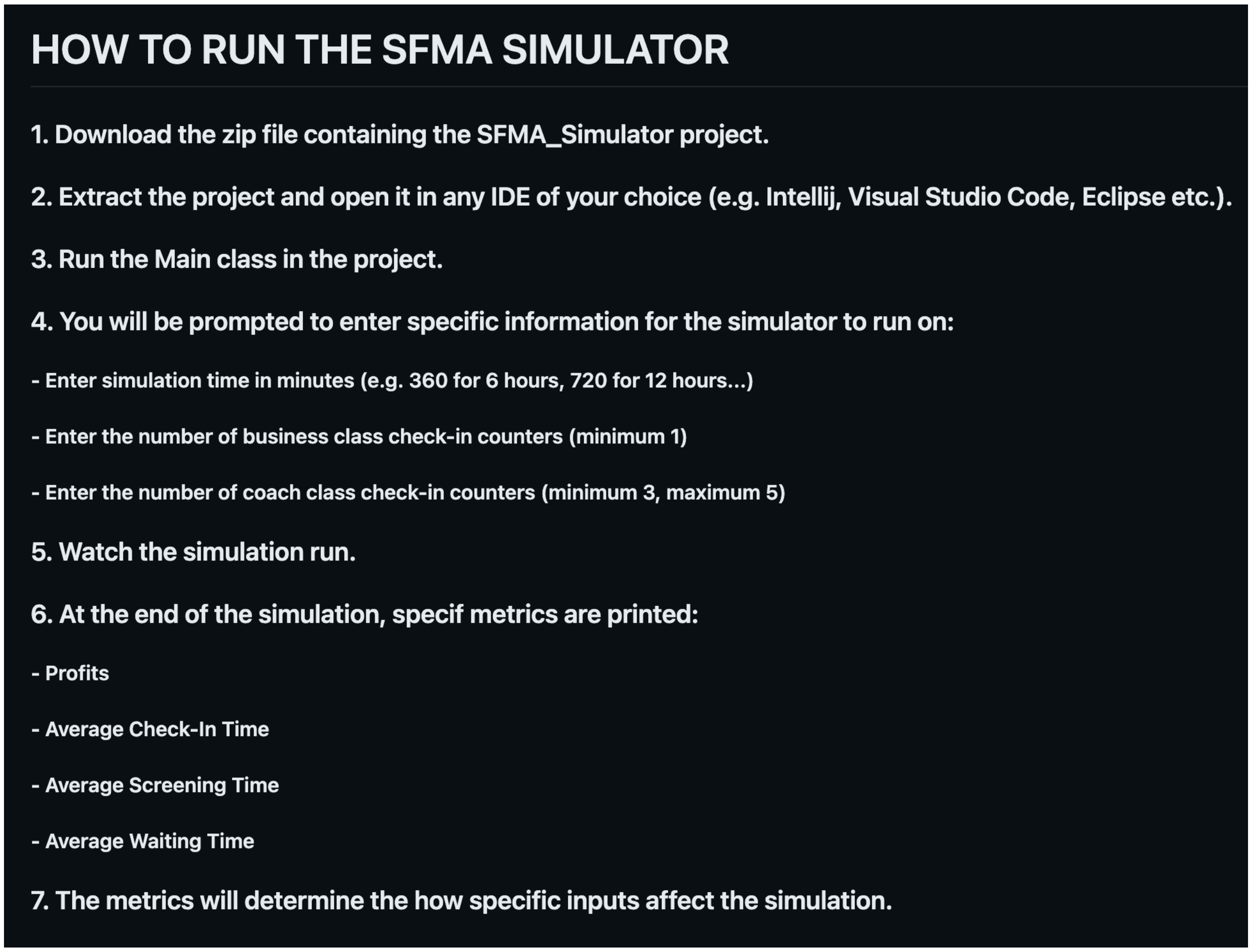
Then, I delved into a detailed analysis of the current operational performance, identifying constraints and areas for improvement. Through this analysis, I gained valuable insights into the dynamics of passenger flow, queue management, and resource allocation within the airport environment.

Additionally, an alternative operating policies was introduced, offering a glimpse into potential strategies for enhancing efficiency and mitigating risks. These alternative policies were carefully crafted to address the specific challenges identified during the analysis phase.

Finally, I justified and evaluated the recommended policy, drawing upon the insights returned from both the simulation results and theoretical considerations. Through this process, I arrived at a well-informed conclusion regarding the most effective course of action for optimizing airport operations.

This final report represents a culmination of rigorous analysis, strategic decision-making, and technical proficiency. It suggests tangible improvements in airport operations, which could ultimately contribute to a more seamless and efficient travel experience for passengers.

APPENDIX A: HOW TO RUN THE SFMA SIMULATOR



HOW TO RUN THE SFMA SIMULATOR

1. Download the zip file containing the SFMA_Simulator project.

2. Extract the project and open it in any IDE of your choice (e.g. Intellij, Visual Studio Code, Eclipse etc.).

3. Run the Main class in the project.

4. You will be prompted to enter specific information for the simulator to run on:

- Enter simulation time in minutes (e.g. 360 for 6 hours, 720 for 12 hours...)

- Enter the number of business class check-in counters (minimum 1)

- Enter the number of coach class check-in counters (minimum 3, maximum 5)

5. Watch the simulation run.

6. At the end of the simulation, specif metrics are printed:

- Profits

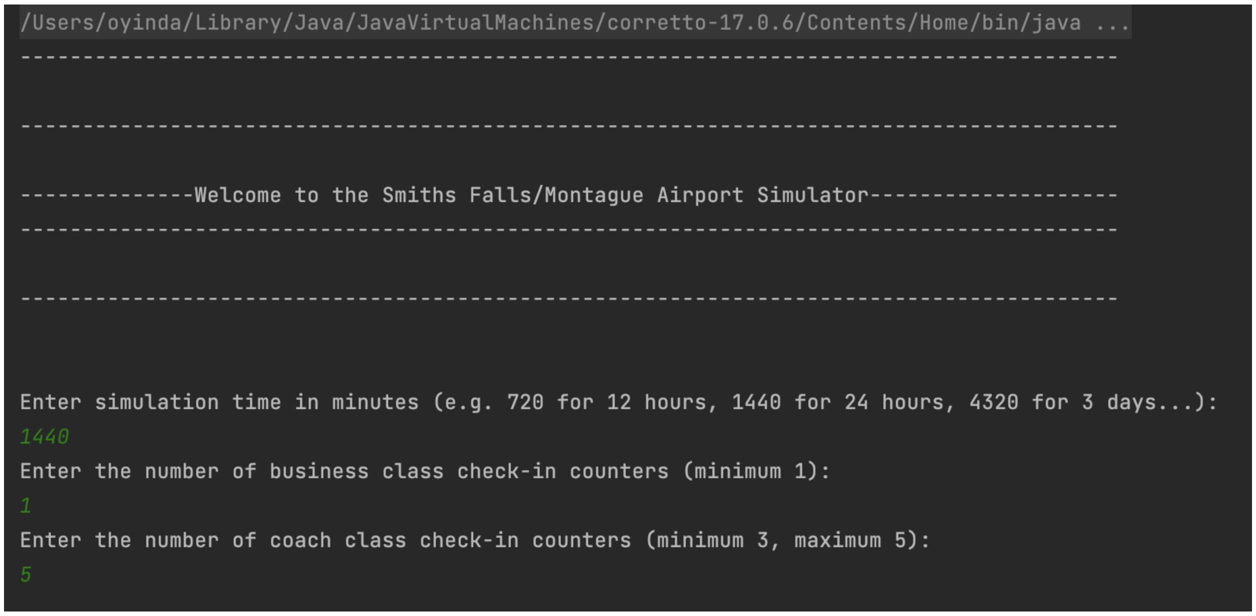
- Average Check-In Time

- Average Screening Time

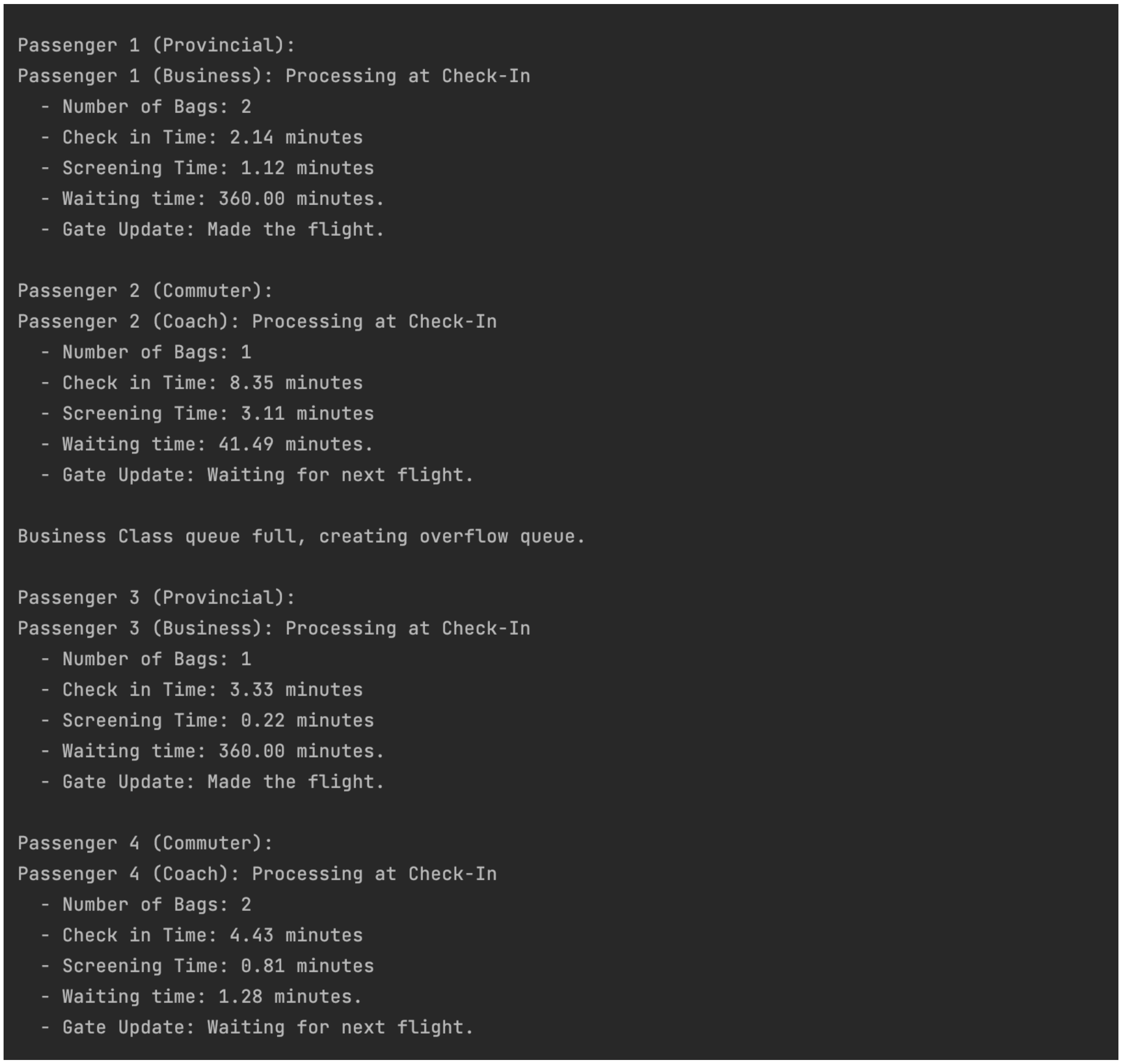
- Average Waiting Time

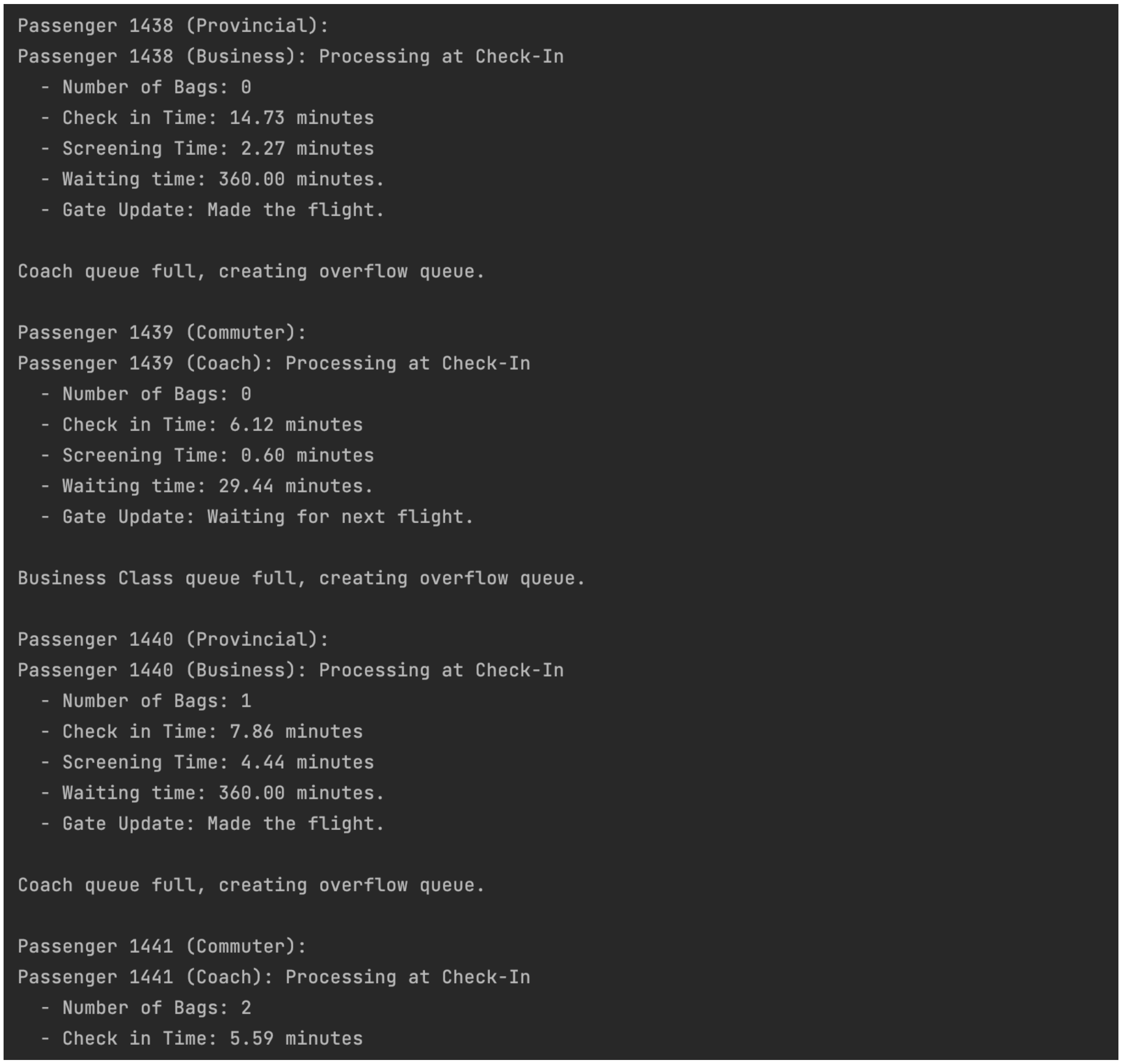
7. The metrics will determine the how specific inputs affect the simulation.

APPENDIX B: SIMULATION EXAMPLE

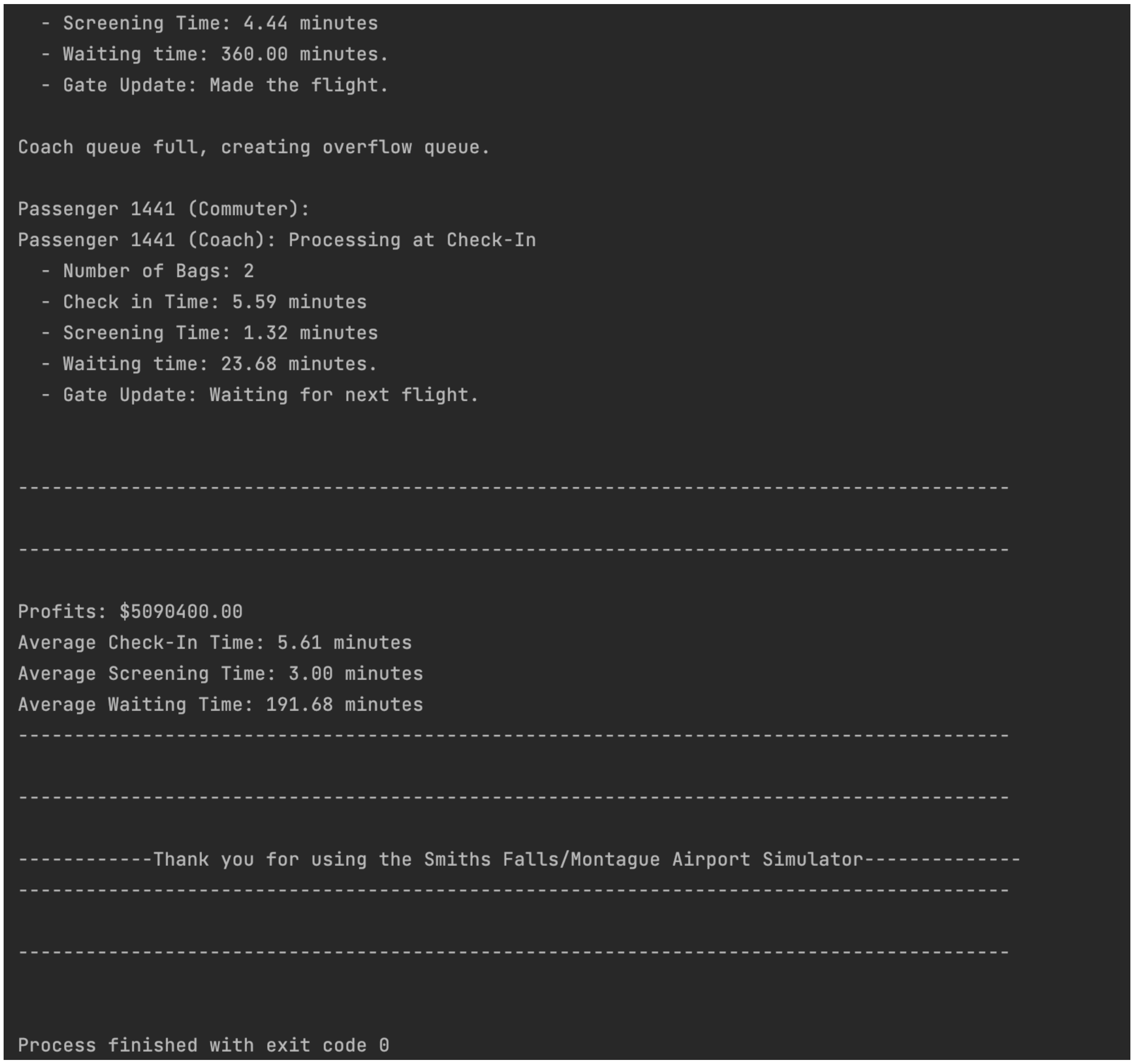


```
/Users/oyinda/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ...
---------------------------------------------------------------------------------------


---------------------------------------------------------------------------------------


--------------Welcome to the Smiths Falls/Montague Airport Simulator--------------------
---------------------------------------------------------------------------------------


---------------------------------------------------------------------------------------



Enter simulation time in minutes (e.g. 720 for 12 hours, 1440 for 24 hours, 4320 for 3 days...):
1440
Enter the number of business class check-in counters (minimum 1):
1
Enter the number of coach class check-in counters (minimum 3, maximum 5):
5
```


```
Passenger 1 (Provincial):
Passenger 1 (Business): Processing at Check-In
  - Number of Bags: 2
  - Check in Time: 2.14 minutes
  - Screening Time: 1.12 minutes
  - Waiting time: 360.00 minutes.
  - Gate Update: Made the flight.

Passenger 2 (Commuter):
Passenger 2 (Coach): Processing at Check-In
  - Number of Bags: 1
  - Check in Time: 8.35 minutes
  - Screening Time: 3.11 minutes
  - Waiting time: 41.49 minutes.
  - Gate Update: Waiting for next flight.

Business Class queue full, creating overflow queue.

Passenger 3 (Provincial):
Passenger 3 (Business): Processing at Check-In
  - Number of Bags: 1
  - Check in Time: 3.33 minutes
  - Screening Time: 0.22 minutes
  - Waiting time: 360.00 minutes.
  - Gate Update: Made the flight.

Passenger 4 (Commuter):
Passenger 4 (Coach): Processing at Check-In
  - Number of Bags: 2
  - Check in Time: 4.43 minutes
  - Screening Time: 0.81 minutes
  - Waiting time: 1.28 minutes.
  - Gate Update: Waiting for next flight.
```



```
Passenger 1438 (Provincial):
Passenger 1438 (Business): Processing at Check-In
  - Number of Bags: 0
  - Check in Time: 14.73 minutes
  - Screening Time: 2.27 minutes
  - Waiting time: 360.00 minutes.
  - Gate Update: Made the flight.

Coach queue full, creating overflow queue.

Passenger 1439 (Commuter):
Passenger 1439 (Coach): Processing at Check-In
  - Number of Bags: 0
  - Check in Time: 6.12 minutes
  - Screening Time: 0.60 minutes
  - Waiting time: 29.44 minutes.
  - Gate Update: Waiting for next flight.

Business Class queue full, creating overflow queue.

Passenger 1440 (Provincial):
Passenger 1440 (Business): Processing at Check-In
  - Number of Bags: 1
  - Check in Time: 7.86 minutes
  - Screening Time: 4.44 minutes
  - Waiting time: 360.00 minutes.
  - Gate Update: Made the flight.

Coach queue full, creating overflow queue.

Passenger 1441 (Commuter):
Passenger 1441 (Coach): Processing at Check-In
  - Number of Bags: 2
  - Check in Time: 5.59 minutes
```



```
  - Screening Time: 4.44 minutes
  - Waiting time: 360.00 minutes.
  - Gate Update: Made the flight.

Coach queue full, creating overflow queue.

Passenger 1441 (Commuter):
Passenger 1441 (Coach): Processing at Check-In
  - Number of Bags: 2
  - Check in Time: 5.59 minutes
  - Screening Time: 1.32 minutes
  - Waiting time: 23.68 minutes.
  - Gate Update: Waiting for next flight.


---------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------

Profits: $5090400.00
Average Check-In Time: 5.61 minutes
Average Screening Time: 3.00 minutes
Average Waiting Time: 191.68 minutes
---------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------

------------Thank you for using the Smiths Falls/Montague Airport Simulator-------------
---------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------


Process finished with exit code 0
```

APPENDIX C: SIMULATION 1 VALUES TABLE

Simulation 1: 1 Hour Simulation with 2 available Business class counters and 4 coach class counters

Check-In Times	Screening Times	Waiting Times
1.39	4.45	360
13.09	5.05	19.5
12.7	0.13	28.54
5.21	2.27	33.55
2.86	0.85	360
6.94	0.39	348.42
3.38	0.35	26.52
5.02	5.57	360
1.96	0.67	6.75
2.92	3.06	30.64
4.96	2.91	50.11
1.79	4.67	360
9.2	2.28	360
4.47	3.1	360
9.26	6.06	360
2.55	7.5	27.46
1.4	0.34	6.17

4.58	0.3	25.07
5.15	2.34	58.82
3.48	0.54	15.42
8.78	11.82	40.05
4.83	10.25	10.36
8.86	0.88	360
4.72	4.62	57.67
5.93	2.76	360
3.12	0.98	360
2.45	1.95	360
4.59	2.18	360
14.22	2.97	3.22
15.75	2.46	44.18
8.43	12.95	360
1.84	4.28	21.13
10.72	0.09	57.1
2.5	1.82	17.6
2.62	1.11	23.25

7.8	0.73	54.65
6.25	11.62	18.98
4.2	0.37	3.22
5.68	2.96	360
8.31	4.45	26.26
15.03	5.05	360
7.41	0.13	57.83
12.95	2.27	7.99
2.68	0.85	360
0.31	0.39	19.5
3.97	0.35	28.54
3.01	5.57	33.55
4.9	0.67	360
7.03	3.06	348.42
3.83	2.91	26.52
6.93	4.67	360
4.9	2.28	6.75
8.03	3.1	30.64

5.12	6.06	50.11
9.79	7.5	360
2.87	0.34	360
1.95	0.3	360
0.54	2.34	360
0.96	0.54	27.46
1.15	11.82	6.17
3.18	10.25	25.07