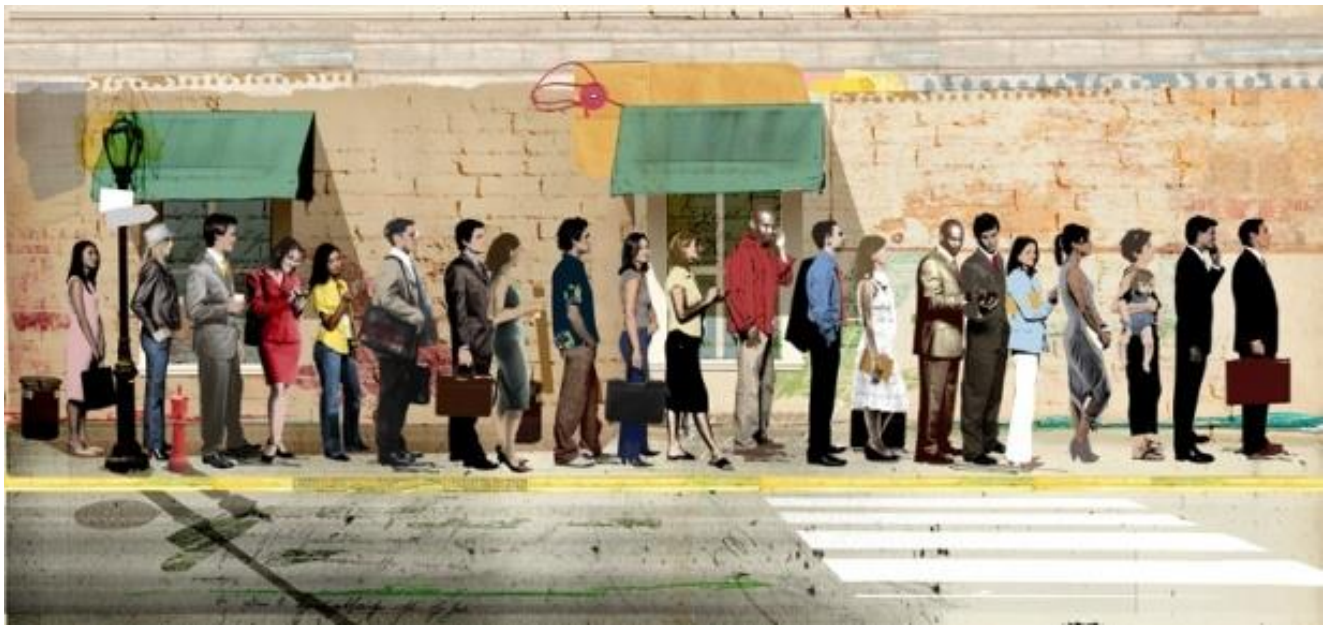
Analysis and Design Document for Smart Queue Management System



Contents

1. Role(s) of Team Member and Assigned Features

Team Members	Feature 1	Feature 2
Yousef Selim 202201255	Simulation Mode	Fairness Monitoring
Mohamed Ehab 202201236	Multiple Queues with merging	Fairness Monitoring
Maysam Asser 202200276	Dynamic Priority Updates	Admin Console
Nadeen Ayman 202300234	Modular Priority Engine	Advanced Reporting and Sorting

2. Detailed Description of Each Feature

2.1. Modular Priority Engine

Description:

This is the core component responsible for calculating and managing the priority of each individual (represented as a delivery object) in the queue. It enables administrators to assign custom weights to priority factors, including urgency, waiting time, and service type. These weights are applied to an individual's attributes to compute a comprehensive priority score, ensuring that the individual with the highest score is served first.

Inputs:

1. Individual ID: Unique identifier (string)
2. Urgency Level: Integer (1-5, 5 being most urgent)
3. Service Type: Enum (URGENT, STANDARD, FRAGILE)
4. Estimated Delivery Time: Integer (minutes)
5. Waiting Time: Integer (minutes, initially 0)
6. urgency_weight: Configurable float (0.5)
7. waiting_time_weight: Configurable float (0.3)
8. service_type_weight: Configurable float (0.2)

9. Service Type Scores: Configurable map/dictionary ('emergency': 10, 'vip': 8, 'regular': 5)

Outputs:

1. Calculated priorityScore for the individual.
2. The individual is inserted into the appropriate priority queue.

Example (User Scenario):

A new individual arrives at the system.

1. User Input:
    1.1. Individual ID: 101
    1.2. Urgency Level: 5 (scale 1-5, 5 being most urgent)
    1.3. Service Type: 'emergency'
2. System-Managed Data:
    2.1. Waiting Time: 0 minutes (initial)
3. Defined Weights (from Admin Console):
    3.1. urgency_weight = 0.5
    3.2. waiting_time_weight = 0.3

3.3. service_type_weight = 0.2

4. Service Type Score (from Admin Console):

4.1. emergency = 10

5. System Processing:

5.1. Retrieve weights from a hash table (or similar configuration storage).

5.2. Calculate priority score using the formula:
Priority Score = (Urgency Level × urgency_weight) + (Waiting Time × waiting_time_weight) + (Service Type Score × service_type_weight)
Priority Score = (5 × 0.5) + (0 × 0.3) + (10 × 0.2)
Priority Score = 2.5 + 0 + 2.0 = 4.5

5.3. Insert individual 101 into the priority queue with score 4.5.

6. System Output:
"Individual ID 101 with priority score 4.5 has been added to the main queue."

2.2 Dynamic Priority Updates

Description: This feature prevents indefinite waiting ("starvation") by periodically updating priority scores based on increased waiting time. Updated scores reposition individuals in the queue, ensuring fairness and accuracy. This builds upon the Modular Priority Engine by re-evaluating and updating the priority of individuals already in the queue.

Inputs:

1. Individual ID
2. Current Waiting Time for an individual in the queue
3. urgency_weight, waiting_time_weight, service_type_weight (from Admin Console)
4. Service Type Scores (from Admin Console)

Outputs:

1. Updated priority Score for the individual
2. Individual repositioned in the priority queue (if score changes)

Example (User Scenario #10):

The individual's priority is re-evaluated after waiting.

1. Initial State:
    1.1. Individual ID: 102
    1.2. Urgency Level (initial): 3
    1.3. Service Type (initial): 'regular'
    1.4. Service Type Score (initial): 5 (for 'regular', from Admin Console)
    1.5. Waiting Time (initial): 10 minutes
    1.6. Initial Score: 5.5 (calculated as: (3 × 0.5) + (10 × 0.3) + (5 × 0.2) = 1.5 + 3.0 + 1.0 = 5.5)
2. System Processing (After 10 more minutes):
    2.1. New Waiting Time: 20 minutes

2.2. Recalculate score using the same priority score formula with the updated waiting time:
New Priority Score = (3 × 0.5) + (20 × 0.3) + (5 × 0.2)
New Priority Score = 1.5 + 6.0 + 1.0 = 8.5

2.3. Update position in the max-heap.

3. System Output:
"Priority for Individual ID 102 updated to 8.5 due to increased waiting time."

2.3 Multiple Queues with Merging

Description:

This feature extends the single-queue system to manage multiple distinct priority queues (VIP, Regular, Emergency). It supports merging or load balancing when operational needs arise, such as redirecting individuals from an empty queue to another service point.

Inputs:

1. New Delivery objects with their DeliveryType
2. Status of existing queues (empty/non-empty)

Outputs:

1. Delivery objects added to their respective queues
2. Individuals from an emptied queue moved to another queue (if merging is triggered

Example (User Scenario #11):

VIP queue empties while the Regular queue has a backlog.

1. Initial State:
    1.1. VIP Queue: [ID=201 (P=8.0), ID=202 (P=7.5)]
    1.2. Regular Queue: [ID=301 (P=6.0), ID=302 (P=5.8), ID=303 (P=5.5)]
2. System Processing:
    2.1. Serve ID=201 and ID=202; VIP queue empties.
    2.2. Merge Regular queue individuals into the VIP queue, preserving their priorities.
3. System Output:
    "The VIP queue is now empty. Redirecting individuals from the regular queue to the VIP service counter."

2.4 Fairness Monitor

Description:

This feature ensures no individual waits excessively by tracking waiting times and flagging those exceeding a defined threshold. When an individual's waiting time surpasses the max_wait_time threshold, their priority is explicitly increased based on the extra time waited. This dynamic boost guarantees that individuals who wait significantly longer receive proportionally higher priority.

Inputs:

1. Individual ID
2. Entry Time (timestamp when joining the queue)
3. Current Time
4. max_wait_time: Configurable integer (25 minutes)
5. boost_multiplier: Configurable float (0.5)
6. Current priorityScore

Outputs:

1. Updated priorityScore (if threshold exceeded)
2. Repositioned the individual in the priority queue
3. System notification about priority boost

Example (User Scenario):

Individual 401 receives a priority boost after excessive wait.

1. Initial State:

1.1. Individual ID: 401

1.2. Entry Time: 10:00 AM

1.3. Current Time: 10:30 AM

1.4. max_wait_time: 25 minutes

1.5. boost_multiplier: 0.5

1.6. Urgency Level: 2

1.7. Service Type: 'regular'

1.8. Service Type Score: 5

1.9. Current Priority Score: 11.0
(Calculation: (2×0.5) + (30×0.3) + (5×0.2) = 1.0 + 9.0 + 1.0 = 11.0)

2. System Processing:

2.1. Calculate waiting time: 30 minutes

2.2. Determine extra wait: 5 minutes (30 - 25 threshold)

2.3. Compute boost: 5 × 0.5 = 2.5

2.4. New Priority Score: 11.0 + 2.5 = 13.5

2.5. Update position in max-heap

3. System Output:
"Warning: Individual ID 401 has waited 30 minutes (5 minutes over threshold).
Priority boosted to 13.5 due to the fairness rule."

2.5 Simulation Mode

Description:

This testing tool simulates queue behavior by generating random individuals with

varying attributes (urgency, service type, estimated time) and displaying real-time queue states. It relies on the proper functioning of the Modular Priority Engine, Dynamic Priority Updates, and Multiple Queues with Merging features.

Inputs:

1. Duration (minutes, ex., 60)
2. Arrival Rate (individuals/minute, ex.,0.5)
3. Service Counters (ex.,3)

Outputs:

1. Snapshots of queue states at various time intervals.
2. Information about new arrivals and processed individuals

Example (User Scenario):

Simulate arrival burst during peak hours.

1. Input Configuration:
    1.1. Duration: 60 minutes
    1.2. Arrival Rate: 0.5 individuals/minute
    1.3. Counters: 3
2. System Output (Snapshot):
    --- Time: 2 minutes —

New Arrival: ID=106 (P=5.0)

Regular Queue: [ID=106 (P=5.0), ID=101 (P=4.8)]

2.6 Advanced Reporting and Sorting

Description:

This feature enables post-service analysis by storing historical data of processed individuals and allowing it to be sorted and filtered based on various criteria (waiting time, service type, service time). This helps identify performance trends and bottlenecks.

Inputs:

1. Filter criteria (service type, date range)
2. Sort criteria (waiting time descending, service time ascending)

Outputs:

Formatted historical reports with relevant metrics

Example (User Scenario):

Generate a report for the regular service queue.

1. Input Configuration:
    1.1. Filter: Service Type = 'regular'
    1.2. Sort: Waiting Time (descending)
2. System Output:

ID=302 | regular | 45 min wait | 12 min service

ID=304 | regular | 38 min wait | 10 min service

2.7 Admin Console

Description:

The Admin Console provides a centralized interface for configuring system parameters and thresholds, thereby controlling the behavior of all other features. This includes setting priority weights, service type scores, fairness thresholds, and simulation parameters.

Inputs:

1. Priority Weights:
    1.1. urgency_weight (float)
    1.2. waiting_time_weight (float)
    1.3. service_type_weight (float)
2. Service Type Scores (map):
    2.1. emergency: 10
    2.2. vip: 8
    2.3. regular: 5
3. Fairness Controls:

3.1. max_wait_time (integer)

3.2. boost_multiplier (float)

4. Simulation Settings:

4.1. duration (minutes)

4.2. arrival_rate (individuals/minute)

4.3. counters (integer)

Outputs:

1. Configuration success notifications
2. Immediate system behavior adjustments

Example (User Scenario):

Modify priority calculation weights.

1. Input:

1.1. urgency_weight: 0.4

1.2. waiting_time_weight: 0.4

1.3. service_type_weight: 0.2

2. System Output:
"Priority weights updated successfully

3. Justifications of Data Structure Selection

3.1. Heaps (MinHeap, MaxHeap)

Selection:

MinHeap and MaxHeap are chosen as the underlying data structures for the PriorityQueue.

Justification (Time Complexity):

1. Insertion (enqueue): O(log N), where N is the number of elements in the heap. A new element is added to the end and then heapified. This is efficient for maintaining the heap property.
2. Extraction (dequeue): O(log N). The root (min or max element) is removed, the last element is moved to the root, and then heapified down. This operation also maintains the heap property efficiently.
3. Peek (peekMin/peekMax): O(1). Accessing the root element is a direct operation.

Justification (Space Complexity):

O(N), where N is the number of elements in the heap. Heaps are typically implemented using arrays or vectors, which store elements contiguously. This provides efficient memory usage.

Suitability:

Heaps are ideal for implementing priority queues because they provide efficient retrieval of the highest (or lowest) priority element and efficient insertion/deletion while maintaining the priority order. The MinHeap and MaxHeap classes are templated, allowing them to be used with any data type that supports the comparison operators (< and >), making them flexible for managing Delivery objects based on different criteria (estimatedDeliveryTime for MinHeap, priorityScore for MaxHeap).

3.2. PriorityQueue

Selection:

The PriorityQueue class acts as an adapter for either a MinHeap or a MaxHeap.

Justification (Time Complexity):

The time complexities for enqueue, dequeue, and peek operations directly inherit from the underlying heap implementation: O(log N) for enqueue and dequeue, and O(1) for peek.

Justification (Space Complexity):

O(N), inherited from the underlying heap.

Suitability:

The PriorityQueue class provides a clean, abstract interface for priority queue operations, decoupling the application logic from the specific heap implementation. This allows for easy switching between min-priority and max-priority behavior without altering the client code, which is beneficial for managing different types of deliveries (if some deliveries need to be prioritized by minimum estimated time and others by maximum urgency score).

3.3. DeliveryManager

Selection:

The DeliveryManager class uses multiple instances of PriorityQueue.

Justification (Time Complexity):

1. addDelivery: O(log N), where N is the number of deliveries in the specific queue being added to. This is because it calls the enqueue method of the underlying PriorityQueue.
2. processNextDelivery: In the current implementation, this involves checking the emptiness of three queues and then calling dequeue on one of them. The dequeue operation is O(log N). In the worst case, it's O(log N) for the largest queue. If merging logic were to involve iterating through elements of one queue to insert into another, the complexity could increase (N * log N for merging N elements into a queue of size N).

Justification (Space Complexity):

O(N_total), where N_total is the total number of deliveries across all queues. Each PriorityQueue instance stores its own set of Delivery objects.

Suitability:

Using multiple PriorityQueue instances within DeliveryManager allows for logical separation and management of different categories of deliveries (URGENT, STANDARD, FRAGILE). This design supports the requirement for "Multiple Queues with Merging" by providing distinct queues that can be individually managed and potentially merged. The current simple prioritization in processNextDelivery can be extended to more sophisticated merging and load-balancing strategies.

4. Implementation Logic (Pseudocode)

4.1. Modular Priority Engine

This pseudocode describes the enhanced delivery class constructor and a mechanism to manage configurable weights and service type scores.

CLASS Delivery:

ATTRIBUTES:

deliveryid: STRING

destination: STRING

priorityScore: INTEGER

deliveryType: ENUM (URGENT, STANDARD, FRAGILE)

estimateDeliveryTime: INTEGER (minutes)

entryTime: DATETIME (for waiting time calculation)

```
// make a function that calculates and assigns priority score for a new delivery

Procedure CalculatePriorityScore(delivery, config):

    // Step 1: Retrieve weights from ConfigurationManager

    urgency_weight ← config.getWeight("urgency")

    waiting_time_weight ← config.getWeight("waiting_time")

    service_type_weight ← config.getWeight("service_type")


    // Step 2: Get score for the delivery's service type

    service_type_score ← config.getServiceTypeScore(delivery.deliveryType)
```

```
    // Step 3: Calculate how long the delivery has waited so far
    current_time ← getCurrentSystemTime()
    seconds_waited ← current_time - delivery.entryTime
    waiting_time ← convertSecondsToMinutes(seconds_waited)

    // Step 4: Set urgency level based on delivery type
    If delivery.deliveryType = URGENT:
        urgency_level ← 5
    Else If delivery.deliveryType = FRAGILE:
        urgency_level ← 4
    Else:
        urgency_level ← 3

    // Step 5: Apply the formula to calculate the priority score
    priority_score ← (urgency_level × urgency_weight) +
                     (waiting_time × waiting_time_weight) +
                     (service_type_score × service_type_weight)

    // Step 6: Update the delivery's priority score
    delivery.priorityScore ← floor(priority_score)

    // Done
EndProcedure
```

```
// When a new individual (delivery) arrives in the system
Procedure HandleNewArrival():
    // 1. Take user input
    Input delivery_id
    Input destination
    Input delivery_type  // (0 = URGENT, 1 = STANDARD, 2 = FRAGILE)
    Input estimated_time

    // 2. Create the delivery object
    new_delivery ← CreateDelivery(delivery_id, destination, delivery_type,
estimated_time)

    // 3. Calculate the priority score
    Call CalculatePriorityScore(new_delivery, config)

    // 4. Add delivery to appropriate priority queue
    Call deliveryManager.addDelivery(new_delivery)

    // 5. Display output to console
    Print "Individual ID", new_delivery.deliveryId,
          "with priority score", new_delivery.priorityScore,
          "has been added to the main queue."
EndProcedure
```

Commentary:

- The Delivery class is extended to include an entryTime attribute to track waiting time. This will be crucial for dynamic priority updates and fairness monitoring.
- A ConfigurationManager class is introduced to centralize the management of configurable parameters (weights, service type scores). This adheres to the principle of separation of concerns and makes the system more flexible.
- The calculatePriorityScore method in Delivery now takes a ConfigurationManager object to retrieve the dynamic weights and scores. This method will be called initially and also during dynamic updates.

4.2. Dynamic Priority Update

This pseudocode describes a mechanism within Delivery Manager to periodically update priority of individuals in the queues.

CLASS DeliveryManager:
ATTRIBUTES:

- urgentDeliveries: PriorityQueue<Delivery>
- standardDeliveries: PriorityQueue<Delivery>
- fragileDeliveries: PriorityQueue<Delivery>
- config_manager: ConfigurationManager
- activeDeliveries: LIST<Delivery>

```
//make a function to update all deliveries' priority scores based on their
current waiting time
```

```
Procedure UpdateAllPriorities():

    tempList ← empty list

    // Step 1: Move all deliveries out of the queues into a temporary list

    While urgentQueue is not empty:

        delivery ← urgentQueue.dequeue()

        Add delivery to tempList


    While standardQueue is not empty:

        delivery ← standardQueue.dequeue()

        Add delivery to tempList


    While fragileQueue is not empty:

        delivery ← fragileQueue.dequeue()

        Add delivery to tempList


    // Step 2: Clear the list of all current deliveries

    Clear deliveryManager.allDeliveries


    // Step 3: Loop over each delivery to update their priority

    For each delivery in tempList:

        // Save old score (optional for logging/debugging)

        old_score ← delivery.priorityScore


        // Recalculate priority score using the modular priority engine

        Call CalculatePriorityScore(delivery, config)
```

```
        // Step 4: Apply fairness rule (optional boost if delivery waited
too long)
        current_time ← getCurrentSystemTime()
        seconds_waited ← current_time - delivery.entryTime
        waiting_time ← convertSecondsToMinutes(seconds_waited)

        If waiting_time > config.getMaxWaitTime():
            extra_wait ← waiting_time - config.getMaxWaitTime()
            boost ← extra_wait × config.getBoostMultiplier()
            delivery.priorityScore ← delivery.priorityScore + floor(boost)

            Print "Priority for Individual ID", delivery.deliveryId,
                  "updated to", delivery.priorityScore,
                  "due to increased waiting time."

        // Step 5: Reinsert delivery into the correct queue
        If delivery.deliveryType = URGENT:
            urgentQueue.enqueue(delivery)
        Else If delivery.deliveryType = STANDARD:
            standardQueue.enqueue(delivery)
        Else:
            fragileQueue.enqueue(delivery)

        // Add back to the list of all deliveries
```

```
        Add delivery to deliveryManager.allDeliveries

    EndFor

EndProcedure
```

Commentary:

- The DeliveryManager would need a method, updatePriorities(), which is called periodically (every minute) by a system timer or scheduler.
- This method iterates through all active deliveries. To efficiently update their position in a heap-based priority queue after a priority change, the most straightforward approach (without a specialized heap update operation) is to remove the item and re-insert it. This implies that the DeliveryManager needs a way to access all active Delivery objects, perhaps by maintaining a separate list or map of all deliveries in addition to the queues.
- The pseudocode for repositionInQueue highlights the complexity of updating elements in a heap. For the provided MinHeap / MaxHeap implementations, a full rebuild of the queue (extract all, update, re-insert all) or a targeted remove-and-re-insert for each updated item would be necessary. The latter requires finding the item, which can be O(N) without additional indexing.

4.3.Multiply Queues with Merging

This pseudocode describes the DeliveryManager's handling of multiple queues and a conceptual mergeQueues operation.

CLASS DeliveryManager:
ATTRIBUTES:

urgentDeliveries: PriorityQueue<Delivery>

standardDeliveries: PriorityQueue<Delivery>

fragileDeliveries: PriorityQueue<Delivery>

```
// Procedure to check and handle queue merging when needed
Procedure MergeQueuesIfNeeded():
    // Step 1: Check if VIP (Urgent) queue is empty
    If urgentQueue is empty:

        // First try to merge from Regular (Standard) queue
        If standardQueue is not empty:
            Print "VIP queue is now empty. Redirecting individuals from regular queue to VIP service counter."

            While standardQueue is not empty:
                delivery ← standardQueue.dequeue()
                urgentQueue.enqueue(delivery)

        // If Regular is also empty, try merging from Fragile
        Else If fragileQueue is not empty:
            Print "VIP queue is now empty. Redirecting individuals from fragile queue to VIP service counter."

            While fragileQueue is not empty:
                delivery ← fragileQueue.dequeue()
                urgentQueue.enqueue(delivery)
```

```
        // Else: All queues are empty, nothing to merge
        Else:
            // No merging possible
            Print "All queues are empty. No individuals to redirect."

    // Step 2: If VIP queue is not empty, do nothing (queues are fine)
    Else:
        // No merging needed at this moment
        Do nothing
EndProcedure
```

Commentary:

- The addDelivery method correctly enqueues deliveries into their respective PriorityQueue instances based on deliveryType.
- The processNextDelivery method is enhanced to include a conceptual mergeQueue call. The merging logic is triggered when a higher-priority queue (urgentDeliveries) becomes empty, and a lower-priority queue (standardDeliveries) has items. The example in the problem statement suggests merging into the now-empty higher-priority queue.
- The mergeQueue method physically moves Delivery objects from the sourceQueue to the destinationQueue by dequeuing from one and enqueuing into the other. This maintains their priorities within the new queue structure.

4.4.Fairness Monitor

This pseudocode describes the FairnessMonitor logic, which would likely be integrated into the periodic updatePriorities call or a separate monitoring thread.

CLASS DeliveryManager:

ATTRIBUTES:

// ... existing attributes

config_manager: ConfigurationManager

```
// Procedure to check for long-waiting individuals and apply fairness
boost if needed

Procedure ApplyFairnessMonitor(individual, max_wait_time,
boost_multiplier):


    // Step 1: Calculate current waiting time

    current_time ← GetCurrentSystemTime()

  waiting_time ← current_time - individual.entry_time


    // Step 2: Check if individual exceeds fairness threshold

    If waiting_time > max_wait_time:


        // Step 3: Calculate extra waiting time

    extra_wait ← waiting_time - max_wait_time


        // Step 4: Calculate fairness boost

        fairness_boost ← extra_wait × boost_multiplier
```

```
        // Step 5: Apply fairness boost to current priority score
        individual.priority_score ← individual.priority_score +
fairness_boost

        // Step 6: Update individual's position in the priority queue
        UpdatePositionInQueue(individual)

        // Step 7: Output warning message
        Print "Warning: Individual ID", individual.ID,
              "has waited", waiting_time, "minutes (",
              extra_wait, "minutes over threshold). Priority boosted to",
              individual.priority_score, "due to fairness rule."

    Else:
        // No boost needed
        Do nothing

EndProcedure
```

Commentary:

- The ConfigurationManager is updated to store maxWaitTime and boostMultiplier.

- The updatePriorities method in DeliveryManager is enhanced to incorporate fairness monitoring logic. For each active delivery, it checks if current_waiting_time exceeds max_wait_time.
- If the threshold is exceeded, extra_waiting_time and fairness_boost are calculated, and the priorityScore is directly increased. This is a direct modification of the priorityScore attribute of the Delivery object.
- After the boost, the Delivery object needs to be repositioned in its respective priority queue to reflect the new higher priority. This again implies the need for an efficient repositionInQueue mechanism or a rebuild.
- If no fairness boost is applied, the regular dynamic priority update (based on waiting time) still occurs.

4.5. Simulation Mode

This pseudocode outlines a SimulateManager class that would interact with the DeliveryManager to run simulations.

CLASS SimulationManager:

ATTRIBUTES:

- deliveryManager: DeliveryManager
- config_manager: ConfigurationManager
- processedDeliveries: LIST<Delivery>
- serviceCounters: INTEGER
- currentSimTime: INTEGER (minutes)

```
PROCEDURE RunSimulation()
```

```
    SET duration ← config.getSimulationDuration()
    SET arrival_rate ← config.getSimulationArrivalRate()
    SET counters ← config.getSimulationCounters()

    PRINT "Starting simulation for", duration, "minutes"
    PRINT "Arrival Rate:", arrival_rate, "individuals per minute"
    PRINT "Counters Available:", counters

    FOR current_time FROM 0 TO duration - 1 DO

        PRINT "--- Time:", current_time, "minutes ---"

        // Step 1: Generate New Arrival
        IF random float between 0 and 1 < arrival_rate THEN
            CALL GenerateRandomDelivery() → new_delivery
            CALL deliveryManager.addDelivery(new_delivery)
            PRINT "New Arrival: ID =", new_delivery.deliveryId, "(P =",
new_delivery.priorityScore, ")"
        ENDIF

        // Step 2: Process Deliveries
        FOR i FROM 1 TO counters DO
            IF deliveryManager.hasDeliveries() THEN
```

```
                CALL deliveryManager.processNextDelivery() →
processed_delivery
                STORE processed_delivery in processedDeliveries list
                PRINT "Processed: ID =", processed_delivery.deliveryId,
"(P =", processed_delivery.priorityScore, ")"
            ELSE
                BREAK
            ENDIF
        ENDFOR

        // Step 3: Update Priorities (including fairness boost if needed)
        CALL deliveryManager.updatePriorities()

        // Step 4: Merge Queues (if necessary)
        CALL deliveryManager.mergeQueues()

        // Step 5: Display Current Queue States
        PRINT "Urgent Queue: ["
        FOR delivery IN urgent queue DO
            PRINT delivery.deliveryId, "(P =", delivery.priorityScore, ")"
        ENDFOR
        PRINT "]"

        PRINT "Standard Queue: ["
        FOR delivery IN standard queue DO
            PRINT delivery.deliveryId, "(P =", delivery.priorityScore, ")"
```

```
        ENDFOR
        PRINT "]"

        PRINT "Fragile Queue: ["
        FOR delivery IN fragile queue DO
            PRINT delivery.deliveryId, "(P =", delivery.priorityScore, ")"
        ENDFOR
        PRINT "]"

    ENDFOR

    PRINT "Simulation finished."

END PROCEDURE
PROCEDURE GenerateRandomDelivery()

    SET id ← "D" + random number between 100 and 999
    SET destination ← "Random Location"
    SET type ← Randomly pick URGENT, STANDARD, or FRAGILE
    SET estimated_time ← Random number between 10 and 129

    CREATE Delivery object with id, destination, type, estimated_time
    RETURN new Delivery

END PROCEDURE
```

Commentary:

- A new SimulationManager class is proposed to encapsulate the simulation logic.
- It takes deliveryManager and ConfigurationManager as dependencies.
- The runSimulation method iterates through a defined duration, simulating time steps. At each step:
    - New random deliveries are generated and added to the DeliveryManager.
    - Deliveries are processed from the queues, simulating service counters.
    - deliveryManager.updatePriorities() is called to trigger dynamic priority updates and fairness monitoring.
    - Snapshots of the queue states are printed.
- Helper functions generateRandomDelivery and calculate_arrivals are placeholders for actual random generation logic.
- ConfigurationManager is extended to store simulation parameters (duration, arrival rate, counters).

4.6. Advanced Reporting and Sorting

This pseudocode describes how historical data would be stored and how reporting and sorting would be performed.

CLASS DeliveryManager:

ATTRIBUTES:

- // ... existing attributes
- processedDeliveriesHistory: LIST<ProcessedDeliveryRecord>

```
PROCEDURE GenerateAdvancedReport(filter_type, sort_key, sort_order)


    INPUT:
        filter_type   ← 'regular', 'urgent', 'fragile'
        sort_key      ←  'waiting_time', 'service_time'
        sort_order    ← 'ascending' or 'descending'


    INITIALIZE:
        report_list ← empty list


    FOR each delivery IN processedDeliveries DO
        IF delivery type matches filter_type THEN
            CALCULATE waiting_time = delivery.serviceStartTime -
delivery.entryTime
```

```
            CALCULATE service_time = delivery.serviceEndTime -
delivery.serviceStartTime

            APPEND (delivery ID, type, waiting_time, service_time) TO
report_list

        ENDIF

    END FOR



    IF sort_key = 'waiting_time' THEN

        SORT report_list BY waiting_time

    ELSE IF sort_key = 'service_time' THEN

        SORT report_list BY service_time

    ENDIF



    IF sort_order = 'descending' THEN

        REVERSE the sorted report_list

    ENDIF



    PRINT "----- Delivery Report -----"

    FOR each item IN report_list DO

        PRINT "ID = " + item.ID + " | " + item.type + " | " +
item.waiting_time + " min | " + item.service_time + " min (Service time)"

    END FOR



END PROCEDURE
```

Commentary:

- The DeliveryManager is extended to maintain a processedDeliveriesHistory list, which stores ProcessedDeliveryRecord objects.
- A ProcessedDeliveryRecord class is introduced to store relevant historical data for each processed delivery.
- A ReportGenerator class is proposed to handle the filtering and sorting of this historical data.
- The generateReport method takes filter_type, sort_by, and sort_order as parameters and prints the formatted report.

This pseudocode describes the AdminConsole class that would interact with the ConfigurationManager.

4.7. AdminConsole:

ATTRIBUTES:

- config_manager: ConfigurationManager
- simulation_manager: SimulationManager // To trigger simulations

```
PROCEDURE DisplayAdminConsole()


    REPEAT UNTIL user chooses Exit:
```

```
        PRINT "--- Admin Console ---"
        PRINT "1. Set Priority Weights"
        PRINT "2. Set Service Type Scores"
        PRINT "3. Set Fairness Thresholds"
        PRINT "4. Set Simulation Parameters"
        PRINT "5. Run Simulation"
        PRINT "6. Add a Delivery"
        PRINT "7. Exit"
        PROMPT "Enter your choice: " → user_choice

        SWITCH user_choice

            CASE 1:
                CALL SetPriorityWeights()

            CASE 2:
                CALL SetServiceTypeScores()

            CASE 3:
                CALL SetFairnessThresholds()

            CASE 4:
                CALL SetSimulationParameters()

            CASE 5:
```

```
                CALL RunSimulation()


            CASE 6:
                CALL AddDeliveryManually()


            CASE 7:
                PRINT "Exiting Admin Console."
                BREAK loop


            DEFAULT:
                PRINT "Invalid choice. Please try again."


        END SWITCH


    END REPEAT


END PROCEDURE
PROCEDURE SetPriorityWeights()
    PROMPT "Enter urgency weight: " → urgency_weight
    PROMPT "Enter waiting time weight: " → waiting_time_weight
    PROMPT "Enter service type weight: " → service_type_weight


    CALL config.setWeight("urgency", urgency_weight)
    CALL config.setWeight("waiting_time", waiting_time_weight)
    CALL config.setWeight("service_type", service_type_weight)
```

```
    PRINT "Priority weights updated successfully."
END PROCEDURE
PROCEDURE SetServiceTypeScores()
    PROMPT "Enter URGENT score: " → urgent_score
    PROMPT "Enter STANDARD score: " → standard_score
    PROMPT "Enter FRAGILE score: " → fragile_score

    CALL config.setServiceTypeScore(URGENT, urgent_score)
    CALL config.setServiceTypeScore(STANDARD, standard_score)
    CALL config.setServiceTypeScore(FRAGILE, fragile_score)

    PRINT "Service type scores updated successfully."
END PROCEDURE
PROCEDURE SetFairnessThresholds()
    PROMPT "Enter max wait time (in minutes): " → max_wait
    PROMPT "Enter boost multiplier: " → boost

    CALL config.setMaxWaitTime(max_wait)
    CALL config.setBoostMultiplier(boost)

    PRINT "Fairness thresholds updated successfully."
END PROCEDURE
PROCEDURE SetSimulationParameters()
    PROMPT "Enter simulation duration (in minutes): " → duration
```

```
    PROMPT "Enter arrival rate (individuals per minute): " → arrival_rate
    PROMPT "Enter number of service counters: " → counters

    CALL config.setSimulationDuration(duration)
    CALL config.setSimulationArrivalRate(arrival_rate)
    CALL config.setSimulationCounters(counters)

    PRINT "Simulation parameters updated successfully."
END PROCEDURE
PROCEDURE RunSimulation()
    CALL simulation_manager.runSimulation()
END PROCEDURE
PROCEDURE AddDeliveryManually()
    PROMPT "Enter Delivery ID: " → id
    PROMPT "Enter Destination: " → destination
    PROMPT "Enter Delivery Type (0=URGENT, 1=STANDARD, 2=FRAGILE): " →
type_index
    PROMPT "Enter Estimated Delivery Time (minutes): " → est_time

    CREATE new Delivery object with id, destination, type_index, and
est_time
    CALL deliveryManager.addDelivery(new_delivery)
END PROCEDURE
```

Commentary:

- An AdminConsole class is proposed to provide a command-line interface for configuring the system.
- It interacts with the ConfigurationManager to set various parameters (priority weights, service type scores, fairness thresholds, and simulation parameters).
- It triggers the SimulationManager to run simulations and the ReportGenerator to generate reports.
- This pseudocode provides a basic menu-driven interface for user interaction, allowing administrators to:
    - Configure system parameters
    - Run simulations
    - Generate reports with customizable filters and sorting