Algorithmic Domain Translation Series

Scenario 1: Airport Boarding Simulation

Boolean Identity Version - based on Amit Amlani's original aisle movement code

Purpose of this document

This document presents airport boarding as a domain translation of the original aisle-shifting algorithm. It deliberately keeps the boolean-style identity logic visible, because the learning value is in seeing how anonymous 1s can still be treated as continuing passengers across repeated scan phases.

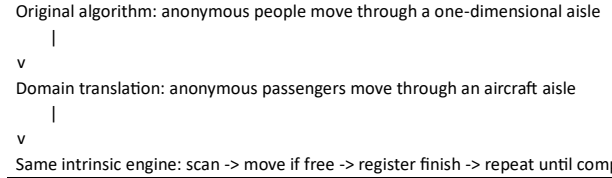
Important scope boundary

The complete passenger journey is described from airport arrival to final seating. However, the actual simulation engine begins only when the passenger enters the aircraft door. All queues, stop-start waves, blockages, and movement statistics described by the code occur inside the aircraft aisle after crossing the aircraft door, not in the wider airport terminal.

Document Structure

1. Series Template: Algorithm -> Domain Translation -> Same Algorithm

The aim of the Algorithmic Domain Translation Series is to show that the same computational structure can appear in very different real-world systems. The code is not merely renamed; it is reinterpreted while preserving its original movement logic, phase resets, finish detection, and identity-tracking behaviour.



Series component	Purpose
Real-world process introduction	Explain what actually happens in the domain before the code is introduced.
Scope boundary	State exactly where the simulation starts and where it ends.
Variable translation	Show how original names map to domain names.
Boolean interpretation	Explain why each boolean still exists in the domain version.

Scenario/test cases	Show what each input pattern is testing.
Annotated code	Keep original fragments inline beside the domain equivalent.
Dataset extensions	Explain how richer real data can be attached later without losing the core engine.

Why the boolean version is intentionally preserved

A cleaner object-oriented model could represent each passenger directly. This document does not do that, because the aim is to stay close to the original code. The original code had to infer identity from 0s and 1s. That is why booleans such as hasPassengerMove and hasPassengerPrevMove remain useful and educational.

2. Airport Boarding Process: From Airport Entry to Final Seat

A real passenger journey contains many stages before the aircraft aisle is reached. The algorithm does not simulate every stage, but the wider process matters because it explains how the aisle state could be produced from a domain dataset.

```
Passenger enters airport terminal
        |
v
Check-in or online check-in confirmation
        |
v
Bag drop, if hold luggage is used
        |
        v
Security screening
        |
v
Passport control, where applicable
        |
        v
Passenger waits at gate
        |
v
Boarding group or row group is called
        |
        v
Boarding pass is scanned
        |
v
Passenger walks through jet bridge or stairs
        |
v
Passenger crosses the aircraft door
        |
v
*** THIS IS WHERE THE SIMULATION ENGINE STARTS ***
        |
v
Passenger occupies an aircraft aisle tile
        |
v
Passenger moves forward if the next aisle tile is free
        |
v
Passenger may be blocked by another passenger storing baggage or reaching a row
        |
v
Passenger reaches assigned row / final seating zone
        |
v
Passenger leaves the aisle and takes final seat before take-off
        |
v
*** THIS IS WHERE THE SIMULATION ENGINE ENDS ***
```

Airport stage	Included in code?	Reason

Airport entrance	Described only	This is before the aisle state exists.
Check-in / bag drop	Described only	Can later influence passenger order or luggage delay.
Security / passport control	Described only	Can later influence arrival distribution at the gate.
Gate waiting and boarding calls	Described only	Can later generate the starting pattern in the aircraft aisle.
Boarding pass scan / jet bridge	Described only	Can later decide how passengers enter the aircraft door.
Aircraft door entry	Simulation start	The passenger becomes a 1 in the aircraft aisle array.
Aircraft aisle movement	Fully simulated	This is the core local movement rule.
Blocking and stop-start behaviour	Fully simulated	These emerge from repeated scans inside the aircraft door.
Final seat / aisle cleared	Simulation end	The passenger is counted as finished for the model.

3. Simulation Scope: The Aircraft Door Boundary

The most important modelling decision is the start boundary. In this version, the simulation does not start when the traveller enters the airport. It starts when the traveller crosses the aircraft door and becomes part of the internal aircraft aisle flow.

Aircraft-door boundary

Everything before the aircraft door is context or dataset preparation. Everything after the aircraft door is simulation. Therefore, when this document refers to queue formation, congestion, blocking, passenger continuation, finish detection, or total moves, those phenomena are all inside the aircraft after entry through the door.

Outside aircraft: terminal, gate, boarding pass scan, jet bridge
|
v
Aircraft door
|
v
Inside aircraft: aisle occupancy array

Example inside-aircraft state: 0 1 1
0 1 0 0 0 1

0 = empty aircraft aisle tile
1 = anonymous passenger currently occupying that tile

4. Original Code to Airport Boarding Mapping

The translation deliberately keeps the original code structure visible. The aim is not to hide the boolean logic inside a Passenger object. The aim is to show a one-to-one relationship between the original aisle algorithm and a realistic aircraft boarding interpretation.

Original concept	Airport translation	Meaning in the simulation
people[]	boardingAisle[]	One-dimensional aircraft aisle after the aircraft door.
1	anonymous passenger	A passenger currently standing in an aircraft aisle tile.
0	empty aisle tile	A space a passenger can move into.
move right	move toward rear seats	Passenger boards deeper into the aircraft.
move left	move toward front door	Reverse scenario such as deplaning or frontdirection movement.
scan row	boarding tick	One discrete simulation phase inside the
		aircraft aisle.

finished moving	finished boarding / reached seat zone	Passenger has reached the modelled completion area.
recordPersonMoved[]	recordPassengerFinishedAt[]	Approximate identity registry for anonymous passengers.
Boolean / variable	**Original role**	**Airport role**
isStartZero	Controls left-to-right or right-to-left scan.	Controls front-door-to-rear or rear-to-front scan.
hasPersonMove	A person has started/currently moved in this scan.	A passenger has started/currently moved in this boarding tick.
hasPersonPrevMove	Detects continuation of a mover.	Detects whether the same anonymous passenger is continuing.
hasPersonFinishedMoving	Marks a finish/stop condition.	Marks passenger reached seat zone or blocked/finished point.
hasPreviousPerson	Previous mover was accounted for.	Previous passenger was accounted for in this tick.
numberScansRow	Counts scan phases.	Counts boarding ticks inside the aircraft.
currentNumberScansRow	Tracks the scan where movement was registered.	Tracks the boarding tick where passenger continuation was registered.

5. Phenomena Covered Inside the Aircraft Door

The following phenomena are not terminal-side airport processes. They are all phenomena inside the aircraft after passengers have crossed the aircraft door and entered the aisle model.

Phenomenon inside aircraft	How it appears in the code
Aisle queue formation	Multiple 1s occupy separated or adjacent tiles and progress only when a 0 exists ahead.
Stop-and-go movement	A passenger moves during one boarding tick, then may stop when the next tile is occupied.
Passenger continuation	hasPassengerPrevMove and recordPassengerFinishedAt help infer whether a moving 1 is the same passenger across scans.
Blocking behind slow passengers	If the next aisle tile is occupied, the passenger cannot move and the chain behind may also stop.
Boundary finish	End-of-scan logic registers a passenger when the model boundary is reached.
Total boarding movement	moves counts the total single-tile movements required by the current scenario.
Average move per passenger	moves divided by numPassengersMoving provides a simple movement efficiency metric.
Direction comparison	The program compares towardRear and towardFront outcomes to identify which movement direction is cheaper for the given pattern.
Compact queue estimate	The analytical report evaluates how many moves are needed to make all passengers adjacent anywhere in the aisle.

Important wording correction

When describing queues and congestion in this document, the queue is not the check-in queue, security queue, or gate queue. It is the aircraft-aisle queue after the passenger has crossed the aircraft door.

6. Test-Case Scenarios

Each scenario starts with a short explanation of what the code is trying to achieve. These are not full airport datasets yet; they are controlled aircraft-aisle patterns designed to test the same movement identity logic as the original code.

Scenario A - Active mixed aisle pattern

Tests a realistic aircraft aisle after several passengers have entered the aircraft door. Some passengers are already close together, some are separated by empty tiles, and the algorithm must preserve movement identity as the scan progresses. int[] boardingAisle = new int[]{0, 1, 1, 0, 1, 0, 0, 0, 1};

Scenario B - Alternating passenger flow

Tests repeated one-tile opportunities. Passengers can advance into gaps, creating a stop-start pattern inside the aircraft aisle. int[] boardingAisle = new int[]{1, 0, 1, 0, 1};

Scenario C - Sparse boarding aisle

Tests passengers that are far apart after entering the aircraft. This shows whether the simulation can handle separated movers without inventing unnecessary interactions. int[] boardingAisle = new int[]{0, 0, 1, 0, 0, 0, 1, 0};

Scenario D - Fully blocked aisle

Tests a saturated aisle. No movement should be possible until some passenger leaves the aisle or reaches a seat in a richer model.

int[] boardingAisle = new int[]{1, 1, 1, 1, 1};

Scenario E - Chain continuation case

Tests the key intrinsic behaviour of the original code: a moving anonymous passenger may continue across scans, and the code must decide whether this is a new passenger or the same one continuing. int[] boardingAisle = new int[]{0, 1, 0, 0, 1, 0, 1};

Scenario F - Single passenger case

Tests the simplest identity case. One passenger enters the aisle, moves to a boundary/seat zone, and completes without interaction.

int[] boardingAisle = new int[]{0, 1, 0};

Scenario G - Empty aircraft aisle

Tests the no-passenger case. The code should avoid false movement and avoid treating empty tiles as passengers. int[] boardingAisle = new int[]{0, 0};

7. Worked Aircraft-Aisle Movement Example

This simplified trace shows how the movement rule behaves once passengers are already inside the aircraft door. It is not modelling check-in or gate waiting; it is modelling the aircraft aisle only.

Initial inside-aircraft aisle state:
0 1 0 1 0 1 0

Boarding tick 1: passengers move into open tiles ahead where possible 0 0 1 0 1 0 1

Boarding tick 2: remaining gaps close toward the target direction 0 0 0 1 0 1 1

Boarding tick 3: movement continues until the finish condition is met 0 0 0 0 1 1 1

The important part is that the code does not simply count 1s. It also tries to interpret movement over time: a 1 that moves in one tick may need to be treated as the same anonymous passenger in the next tick. This is why the original boolean structure remains meaningful in the airport version.

8. Annotated Java Code

The code below is the airport boarding translation using boolean identity logic. Original fragments are kept inline where possible so the relationship to the original program remains visible.

Reading guide for the code

Focus on the phase reset inside beginBoardingMove. That is the core lesson: each boarding tick resets temporary movement flags, then the scan applies the local movement rule. This is what makes the program resemble a discrete simulation rather than a simple one-pass array manipulation.

Code block 1

```
/*
================================================================================
AIRPORT BOARDING SIMULATION - BOOLEAN IDENTITY VERSION
================================================================================ Purpose of this
version:
- Keep Amit's original aisle-shifting structure very close.
- Convert the story to airport boarding.
- Preserve the important boolean-style identity logic inline.
- Keep original code fragments commented beside the airport equivalent where possible.

Core mapping:
   people[]              -> boardingAisle[]
   1                  -> anonymous passenger in aisle
   0                  -> empty aisle tile     person
-> passenger
   move right            -> board toward rear seats     move left             ->
deplane / move toward front door     scan row              -> boarding tick /
boarding phase     finished moving         -> reached final packed boarding
zone     recordPersonMoved[]     -> recordPassengerFinishedAt[]

Important limitation:
This version intentionally keeps passengers anonymous as 1s.
That is why the boolean logic remains important: identity must be inferred through scans.
================================================================================
*/ import java.util.Arrays;

class Main
{
   public static void main(String[] args)
   {
      AirportBoardingBooleanIdentity sim = new AirportBoardingBooleanIdentity();        sim.boardLeftOrRightVerdict();
      sim.reportMinimumAdjacentBoardingZoneMoves();
   }
}

class AirportBoardingBooleanIdentity
{
   // ============================================================
   // TEST CASES AT TOP - airport boarding equivalents
   // ============================================================
   // 0 = empty aisle tile
   // 1 = anonymous passenger standing in aisle
   // The active test case below is intentionally the same shape as your original.

   int[] boardingAisle = new int[]{0, 1, 1, 0, 1, 0, 0, 0, 1};

   // Original equivalent:
   // int[] people = new int[]{0, 1, 1, 0, 1,0,0,0,1};

   // Additional boarding test cases:
   // int[] boardingAisle = new int[]{1, 0, 1, 0, 1};           // alternating passengers
   // int[] boardingAisle = new int[]{0, 0, 1, 0, 0, 0, 1, 0};    // sparse boarding aisle
   // int[] boardingAisle = new int[]{1, 1, 1, 1, 1};           // fully blocked aisle
   // int[] boardingAisle = new int[]{0, 1, 0, 0, 1, 0, 1};       // chain continuation case
   // int[] boardingAisle = new int[]{0, 1, 0, 0, 1, 0};         // two passengers, open aisle
   // int[] boardingAisle = new int[]{0, 0};                  // empty aisle     // int[]
boardingAisle = new int[]{0, 1, 0};               // one passenger
```

```
   int aisleLength = boardingAisle.length;     int
numberPassengers = countPassengers();

   // ========================================================
   // ORIGINAL STATE VARIABLES, AIRPORT RENAMED
   // ========================================================

   boolean isStartAtFrontDoor = false;     //
ORIGINAL:
   // boolean isStartZero=false;     //
AIRPORT MEANING:
   // true  => scan from front door / left side toward rear / right side.     // false
=> scan from rear / right side toward front door / left side.

   boolean hasMultiplePassengers = false;     //
ORIGINAL:
   // boolean hasMultiplePeople=false;     //
AIRPORT MEANING:
   // Set true if the proximity scan finds two passengers on one side.

   boolean hasMinimumSinglePassenger = false;     //
ORIGINAL:
   // boolean hasMinimumSinglePerson=false;     //
AIRPORT MEANING:
   // Set true if at least one passenger exists anywhere in the aisle.

   int count = 0;     int
distanceFromFront = 0;     int
distanceFromRear = 0;

   // ORIGINAL:
   // int distanceLeft=0;     //
int distanceRight=0;

   int passengerPositions[] = new int[2];     int
singlePassengerLocation;

   // ORIGINAL:
   // int pos[]=new int[2];     // int

singlePersonLocation;     double

averageMoves;

   int totalMovesTowardRear = 0;     int
totalMovesTowardFront = 0;     int
totalPassengersMovedTowardRear = 0;     int
totalPassengersMovedTowardFront = 0;     double
averageMovesTowardRear = Double.NaN;     double
averageMovesTowardFront = Double.NaN;

   // ORIGINAL EQUIVALENT:
   // int totalMovesRight = 0;
   // int totalMovesLeft  = 0;
   // int totalPeopleMovedRight = 0;
   // int totalPeopleMovedLeft  = 0;
   // double averageMovesRight = Double.NaN;     //
double averageMovesLeft  = Double.NaN;

   public AirportBoardingBooleanIdentity()
   {
      int farEnd;
int start = 0;
String
boardingDirection;

      System.out.println("AIRPORT BOARDING BOOLEAN IDENTITY SIMULATION");
      System.out.println("Aisle length is: " + aisleLength);
      System.out.println("Number passengers is: " + numberPassengers);
      System.out.println("Original boarding aisle: " + Arrays.toString(boardingAisle));        System
```

Code block 2

```
.out.println("0 = empty aisle tile, 1 = anonymous passenger");        System.out.println();

        System.out.println("A proximity technique from front and rear estimates a likely boarding direction.");
        System.out.println("IMPORTANT: this heuristic is not always correct, so full simulations are compared.");

        System.out.println("\n**** Calculating proximity of two passengers from FRONT DOOR side ****");        farEnd =
aisleLength;

        if (start == 0)
        {
           isStartAtFrontDoor = true;
           boardingDirection = "towardRear";
        }
else
        {
           boardingDirection = "towardFront";
        }

        scanSideToSide(start, farEnd);
        System.out.println("*************** BOARDING ALL PASSENGERS: " + boardingDirection);
beginBoardingMove(start, farEnd, boardingDirection);        isStartAtFrontDoor = false;

        System.out.println("\n**** Calculating proximity of two passengers from REAR side ****");        start =
aisleLength - 1;        farEnd = 0;

        if (start == 0)
        {
           isStartAtFrontDoor = true;
           boardingDirection = "towardRear";
        }
else
        {
           boardingDirection = "towardFront";
        }

        scanSideToSide(start, farEnd);
        System.out.println("*************** BOARDING ALL PASSENGERS: " + boardingDirection);        beginBoardingMove(start, farEnd,
boardingDirection);
    }

    public int countPassengers()
    {        int total =
0;
        for (int i = 0; i < aisleLength; i++)
        {
           if (boardingAisle[i] == 1)
           {
              total++;
           }
        }
        return total;
    }

    public void scanSideToSide(int start, int otherEnd)
    {
        // ORIGINAL:
        // public void sideToSide(int start, int otherEnd)
        for (int i = start;
             isStartAtFrontDoor ? i < otherEnd : i >= otherEnd;            i =
isStartAtFrontDoor ? i + 1 : i - 1)
        {
           System.out.println("aisle tile " + i + " contains: " + boardingAisle[i]);

           if (count == 2)
           {
break;
           }

           if (boardingAisle[i] == 1)
           {
```

```
            passengerPositions[count] = i;
count++;

            if (count == 1)
            {
               singlePassengerLocation = i;
               hasMinimumSinglePassenger = true;
            }
         }
      }

      if (count == 2)
      {
         if (start == 0)
         {
            distanceFromFront = passengerPositions[1] - passengerPositions[0] - 1;              System.out.println("Distance from
front-side pair is: " + distanceFromFront + "\n");
         }
else
         {
            distanceFromRear = passengerPositions[0] - passengerPositions[1] - 1;              System.out.println("Distance from
rear-side pair is: " + distanceFromRear + "\n");
         }
         hasMultiplePassengers = true;
      }
      else if (count == 1)
      {
         System.out.println("One passenger in aisle: location " + singlePassengerLocation);
      }
else
      {
         System.out.println("Empty aisle");        }

      count = 0;
   }

   public void boardLeftOrRightVerdict()
   {
      // ORIGINAL:
      // public void leftOrRight()

      System.out.println("\nAMIT AMLANI ORIGINAL-STYLE BOARDING VERDICT");
      System.out.println("Reason: only inspects the first two passengers from each side.");

      if (hasMinimumSinglePassenger)
      {
         if (distanceFromFront > distanceFromRear)
         {
            System.out.println("\n******VERDICT: Board passengers toward rear\n");
         }
         else if (distanceFromRear > distanceFromFront)
         {
            System.out.println("\n******VERDICT: Move passengers toward front\n");
         }
else
         {
            if (distanceFromFront == 0 && distanceFromRear == 0 && !hasMultiplePassengers)
            {
               if ((aisleLength - 1 - singlePassengerLocation) < singlePassengerLocation)
               {
                  System.out.println("\n******VERDICT: Move single passenger toward rear\n");
               }
               else if (singlePassengerLocation < (aisleLength - 1) - singlePassengerLocation)
               {
                  System.out.println("\n******VERDICT: Move single passenger toward front\n");
               }
else                  {
                  System.out.println("\n******VERDICT: no difference moving single passenger\n");
               }
}              else
{
               System.out.println("\n******VERDICT: no difference moving front or rear\n");
            }
```

```
        }
}

    System.out.println(
```

Code block 3

```
"------------------------------------------------------------");
    System.out.println("SIMULATION VERDICT => compares actual full boarding outcomes");

    if (!hasMinimumSinglePassenger)
    {
return;
    }

    if (totalMovesTowardRear < totalMovesTowardFront)
    {
      System.out.println("\n******VERDICT: Board passengers toward rear\n");
    }
    else if (totalMovesTowardFront < totalMovesTowardRear)
    {
      System.out.println("\n******VERDICT: Move passengers toward front\n");
    }
else
    {
      if (!Double.isNaN(averageMovesTowardRear) && !Double.isNaN(averageMovesTowardFront))
      {
        if (averageMovesTowardRear < averageMovesTowardFront)
        {
          System.out.println("\n******VERDICT: Board passengers toward rear\n");
        }
        else if (averageMovesTowardFront < averageMovesTowardRear)
        {
          System.out.println("\n******VERDICT: Move passengers toward front\n");
        }
else            {
        System.out.println("\n******VERDICT: no difference moving front or rear\n");
        }
}       else
{
        System.out.println("\n******VERDICT: no difference moving front or rear\n");
      }
}
    System.out.println("------------------------------------------------------------");
  }

  public boolean checkAllPassengersFinished(boolean hasConfirmedFinishedBoarding, String boardingDirection)
  {
    // ORIGINAL:
    // public boolean checkHasFinished(boolean hasConfirmedFinishedMoving, String directionMove)      int

interestedLocation;      for (int i = 0; i < numberPassengers; i++)

    {
      if ("towardRear".equals(boardingDirection))
      {
        interestedLocation = aisleLength - (i + 1);
      }
else
      {
        interestedLocation = i;
      }

      if (!(boardingAisle[interestedLocation] == 1))
      {
        hasConfirmedFinishedBoarding = false;
      }
    }
    return hasConfirmedFinishedBoarding;
  }

  public void beginBoardingMove(int start, int otherEnd, String boardingDirection)
```

```
  {
     // ORIGINAL:
     // public void beginMove(int start, int otherEnd, String directionMove)

     boolean hasFinishedBoarding = true;        // ORIGINAL:
boolean hasFinishedMoving=true;

     boolean hasPassengerPrevMove = false;
     // ORIGINAL: boolean hasPersonPrevMove=false;
     // Meaning: used to identify whether this is a continuation of a passenger movement in this scan.

     boolean hasPreviousPassenger = false;
     // ORIGINAL: boolean hasPreviousPerson=false;
     // Meaning: previous anonymous passenger has just been registered as finished.

     boolean hasPassengerMove = false;
     // ORIGINAL: boolean hasPersonMove=false;
     // Meaning: a passenger has started or is currently moving during this scan.

     boolean hasPassengerFinishedBoarding = true;
     // ORIGINAL: boolean hasPersonFinishedMoving=true;
     // Meaning: current anonymous passenger has reached a stop/finish condition.

     int numPassengersMoving = 0;       //
ORIGINAL: int numPeopleMoving=0;

     int moves = 0;
     int[] original = new int[aisleLength];       int k
= 0;

     System.arraycopy(boardingAisle, 0, original, 0, aisleLength);

     int[] recordPassengerFinishedAt = new int[50];
     // ORIGINAL: int [] recordPersonMoved = new int[50];
     // Airport meaning: because passengers are anonymous 1s, this approximates identity by recording finish positions.

     int numberBoardingTicks = 0;
     // ORIGINAL: int numberScansRow=0;
     // Airport meaning: each scan is a boarding tick / simulation phase.

     int currentNumberBoardingTicks = 0;       //
ORIGINAL: int currentNumberScansRow=0;

     int indexMove;
int finishPoint = 0;

     do
     {
        numberBoardingTicks++;          hasFinishedBoarding =
true;

        // ========================================================
        // PHASE RESET - this is the key connection to your code
        // ========================================================
        // ORIGINAL FIX:
        // hasPersonPrevMove = false;
        // hasPreviousPerson = false;
        // hasPersonMove = false;
        // hasPersonFinishedMoving = true;
        //
        // AIRPORT EQUIVALENT:
        hasPassengerPrevMove = false;
hasPreviousPassenger = false;          hasPassengerMove = false;
hasPassengerFinishedBoarding = true;

        System.out.println("\nBOARDING TICK " + numberBoardingTicks + " START: " + Arrays.toString(boardingAisle));

        for (int i = start
```

Code block 4

```
;
```

```
            isStartAtFrontDoor ? i < (otherEnd - 1) : i > otherEnd;                i
= isStartAtFrontDoor ? i + 1 : i - 1)
        {
           if (boardingDirection.equals("towardFront"))
           {
              indexMove = i - 1;
           }
else
           {
              indexMove = i + 1;
           }

           if ((boardingAisle[indexMove] == 0) && (boardingAisle[i] == 1))
           {
              hasPassengerMove = true;

              if (!hasPassengerPrevMove && hasPassengerMove)
              {
                 if (i == (aisleLength - 1))
                 {
k = 1;
                 }

                 for (int m = 0; m < k; m++)
                 {
                    if (i == recordPassengerFinishedAt[m])
                    {
                       System.out.println("\n1START POS / SAME PASSENGER CONTINUES: " + Arrays.toString(boardingAisle));
                       System.out.println("Same anonymous passenger continuing from aisle tile: " + i);                          hasPassengerFinishedBoarding =
false;
                       currentNumberBoardingTicks = numberBoardingTicks;
break;
                    }
                    else if (m == (k - 1))
                    {
                       System.out.println("\n2START POS / NEW PASSENGER STARTS: " + Arrays.toString(boardingAisle));
System.out.println("New anonymous passenger has commenced movement at aisle tile: " + i);                          hasPassengerFinishedBoarding = false;
                       numPassengersMoving++;
                       System.out.println("*************************NUMBER PASSENGERS MOVING: " + numPassengersMoving);
currentNumberBoardingTicks = numberBoardingTicks;
                    }
                 }

                 if (k == 0 && numberPassengers != 0)
                 {
                    System.out.println("\n4START POS / FIRST PASSENGER STARTS: " + Arrays.toString(boardingAisle));
                    System.out.println("New anonymous passenger has commenced movement at aisle tile: " + i);
                    System.out.println("*************************NUMBER PASSENGERS MOVING: " + (numPassengersMoving + 1));

                    hasPassengerFinishedBoarding = false;
numPassengersMoving++;
                    currentNumberBoardingTicks = numberBoardingTicks;
                 }
}

              // ORIGINAL MOVE:
              // people[i]=0;
              // people[indexMove]=1;
              // moves++;
              // System.out.println("             " + Arrays.toString(people));
              //
              // AIRPORT MOVE:
boardingAisle[i] = 0;
boardingAisle[indexMove] = 1;                 moves++;
              System.out.println("            " + Arrays.toString(boardingAisle));

              hasPassengerPrevMove = true;
           }
           else if (hasPassengerMove && boardingAisle[i] != 0)
           {
              if (numberBoardingTicks > currentNumberBoardingTicks && currentNumberBoardingTicks != 0)
              {
                 currentNumberBoardingTicks = numberBoardingTicks;
```

```
                if ("towardRear".equals(boardingDirection))
                {
                  finishPoint = aisleLength - 1;
                }
                else if ("towardFront".equals(boardingDirection))
                {
                  finishPoint = 0;
                }

                System.out.println("PREVIOUS PASSENGER FINISHED / BLOCKED AT POSITION: " + finishPoint);                hasPassengerFinishedBoarding
= true;
                System.out.println("REGISTERING UNIQUE ID APPROXIMATION FOR PREVIOUS PASSENGER: " + finishPoint);
recordPassengerFinishedAt[k] = 0;                k++;
                hasPreviousPassenger = true;
              }
else                {
                System.out.println("PASSENGER FINISHED / BLOCKED AT POSITION: " + i);                hasPassengerFinishedBoarding = true;
              }

              hasPassengerMove = false;
hasPassengerPrevMove = false;

              if (numberBoardingTicks > currentNumberBoardingTicks && currentNumberBoardingTicks != 0)
              {
System.out.println("REG
```

Code block 5

```
ISTERING UNIQUE ID APPROXIMATION FOR PASSENGER: " + 0);
              }
else
              {
                if (!hasPassengerFinishedBoarding)
                {
                  System.out.println("PASSENGER FINISHED / BLOCKED AT POSITION: " + i);
                }

                if (hasPreviousPassenger)
                {
                  System.out.println("PASSENGER FINISHED / BLOCKED AT POSITION: " + i);                hasPreviousPassenger =
false;
                }

                System.out.println("REGISTERING UNIQUE ID APPROXIMATION FOR PASSENGER: " + i);
              }
              recordPassengerFinishedAt[k] = i;
k++;
            }
          }

          // ====================================================
          // END-OF-SCAN FINISH CHECK
          // ====================================================
          // ORIGINAL:
          // if (hasPersonMove) { int endPoint = directionMove.equals("right") ? (length - 1) : 0; ... }

          if (hasPassengerMove)
          {
            int endPoint = boardingDirection.equals("towardRear") ? (aisleLength - 1) : 0;

            if (boardingAisle[endPoint] == 1)
            {
              System.out.println("PASSENGER FINISHED / REACHED BOARDING BOUNDARY AT POSITION: " + endPoint);
System.out.println("REGISTERING UNIQUE ID APPROXIMATION FOR PASSENGER: " + endPoint);

              recordPassengerFinishedAt[k] = endPoint;
k++;
            }

            hasPassengerMove = false;
            hasPassengerPrevMove = false;
          }
```

```
        hasFinishedBoarding = checkAllPassengersFinished(hasFinishedBoarding, boardingDirection);

    } while (!hasFinishedBoarding);

    System.out.println("Number passengers moved: " + numPassengersMoving);
    k = 0;

    System.out.println("\n\n------------------------------------------------------------");
    System.out.println("Total number of moves " + boardingDirection + ": " + moves);
    System.out.println("FINISHED BOARDING AISLE: " + Arrays.toString(boardingAisle));       System.out.println("Number passengers
moving for average: " + numPassengersMoving);

    if (numPassengersMoving == 0)
    {
      averageMoves = 0.0;
    }
else
    {
      averageMoves = (double)moves / (double)numPassengersMoving;
    }

    System.out.println("AVERAGE MOVE PER PASSENGER: " + averageMoves);
    System.out.println("------------------------------------------------------------");

    if ("towardRear".equals(boardingDirection))
    {
      totalMovesTowardRear = moves;
      totalPassengersMovedTowardRear = numPassengersMoving;          averageMovesTowardRear =
averageMoves;
    }
else
    {
      totalMovesTowardFront = moves;
      totalPassengersMovedTowardFront = numPassengersMoving;          averageMovesTowardFront =
averageMoves;
    }

    System.arraycopy(original, 0, boardingAisle, 0, aisleLength);    }

  public void reportMinimumAdjacentBoardingZoneMoves()
  {
    // ORIGINAL:
    // reportMinimumAdjacencyMoves()       //
Airport equivalent:
    // What is the minimum number of single-tile moves to make passengers form one compact boarding queue?

    System.out.println("\n============================================================");
    System.out.println("MINIMUM MOVES TO MAKE ALL PASSENGERS ADJACENT ANYWHERE");
System.out.println("============================================================");

    if (numberPassengers <= 1)
    {
      System.out.println("Trivial case: numberPassengers = " + numberPassengers + ". Already adjacent.");
      System.out.println("Original aisle: " + Arrays.toString(boardingAisle));          return;
    }

    int[] passengerOnePositions = new int[numberPassengers];
    int idx = 0;

    for (int i = 0; i < aisleLength; i++)
    {
      if (boardingAisle[i] == 1)
      {
        passengerOnePositions[idx] = i;
idx++;
        if (idx == numberPassengers) break;
      }
    }

    System.out.println("Original aisle: " + Arrays.toString(boardingAisle));
    System.out.println("Passenger positions left-to-right: " + Arrays.toString(passengerOnePositions));       System.out.println("Evaluating compact
boarding blocks of size: " + numberPassengers + "\n");
```

```
        int bestStart = -1;
        int bestMoves = Integer.MAX_VALUE;

        for (int s = 0; s <= (aisleLength - numberPassengers); s++)
        {
            int total = 0;
            StringBuilder details = new StringBuilder();

            for (int j = 0; j < numberPassengers; j++)
```

Code block 6

```
 {
                int from = passengerOnePositions[j];
  int to = s + j;
                int d = Math.abs(from - to);
  total += d;

                if (j > 0) details.append(" | ");
                details.append("Passenger").append(j).append(": ").append(from).append("->").append(to).append(" (").append(d).append(")");
            }

            System.out.println("Block start s=" + s + " targets [" + s + "..." + (s + numberPassengers - 1) + "]"
  + "  TOTAL MOVES=" + total);           System.out.println("  " + details);

            if (total < bestMoves)
            {
                bestMoves = total;
                bestStart = s;
            }
        }

        System.out.println("\n>>> BEST COMPACT BOARDING BLOCK START = " + bestStart
                    + "  targets [" + bestStart + "..." + (bestStart + numberPassengers - 1) + "]");
        System.out.println(">>> MINIMUM TOTAL MOVES = " + bestMoves);

        int[] target = new int[aisleLength];
        for (int i = bestStart; i < bestStart + numberPassengers; i++)
        {
            target[i] = 1;
        }

        System.out.println("Target arrangement: " + Arrays.toString(target));
        System.out.println("============================================================\n");
    } }
```

9. Dataset Extensions for a More Realistic Airport Model

The current version uses a binary aisle array because that is closest to the original code. A real airport boarding dataset could still be attached around the same movement engine. The key is that the richer data would create the initial aisle state or add delay conditions, while the local movement engine remains recognisable.

Dataset feature	How it could affect the simulation
Aircraft type	Changes aisle length, number of rows, and possibly number of doors.
Seat assignment	Defines where each passenger should leave the aisle and sit.
Boarding group	Determines the order in which passengers reach the aircraft door.
Cabin baggage size	Adds delay when a passenger reaches the row and stores luggage.
Passenger walking speed	Changes how often a passenger can move during a boarding tick.
Families or groups	Creates grouped movement or extra blocking around the same row.
Assistance passengers	Adds slower movement or additional stopping conditions.
Overhead locker capacity	Can cause passengers to stop, search, or move further down the aisle.

Boarding through rear and front doors	Creates two entry boundaries and two interacting aisle flows.
Late arrivals	Adds new passengers to the aircraft door after the simulation has already started.

Why this scales naturally

The present code answers a narrow but important question: how anonymous passengers move and preserve inferred identity inside the aircraft aisle. Richer airport data would not invalidate the logic. It would mainly decide who enters, when they enter, how long they pause, and where they finish.

10. Closing Interpretation

This airport boarding simulation is a strong first entry in the domain translation series because it is close to the original aisle code. The aircraft aisle is naturally one-dimensional, passengers really do block one another, and identity must be preserved across repeated movement phases. The most important modelling boundary is the aircraft door: before that point, the airport process supplies context and data; after that point, the simulation begins.

For learning purposes, the boolean version is valuable precisely because it does not jump straight into a Passenger object. It shows the difficult intermediate step: preserving identity when the data structure only contains anonymous occupancy values. That is very close to the intrinsics of the original code.

11. Why the Simulation Works

The airport boarding simulation deliberately preserves the boolean identity structure of the original algorithm instead of replacing it with a conventional object-oriented passenger model. This decision is educational rather than technical. By keeping anonymous occupancy values (0 and 1), the reader can observe how movement, continuation and completion are inferred through repeated boarding ticks. The algorithm therefore exposes the intermediate reasoning process that would normally be hidden behind richer data structures.

12. Why the Aircraft Door?

The aircraft door forms the modelling boundary. Activities before the door—check-in, security, gate waiting and boarding groups—provide context and future datasets. The simulation itself begins only when a passenger occupies the aircraft aisle and ends when that passenger leaves the aisle to enter the assigned seat.

13. Key Takeaway

The computational engine remains intentionally close to the original implementation while the surrounding interpretation changes completely. This demonstrates that algorithms can often be translated into new domains without altering their intrinsic behaviour.

14. Decision Support and Operational Applications

Although intentionally simple, the simulation illustrates how deterministic movement models can support operational decision making. Different boarding strategies can be compared, congestion points identified and alternative cabin layouts evaluated before changes are introduced in practice.

Potential applications include boarding policy comparison, cabin design studies, crew training, educational demonstrations, operational research and digital-twin style experimentation.

15. Chapter Summary

A simple movement engine becomes significantly more valuable when interpreted as a decision-support framework rather than merely an array-processing algorithm.

16. Towards a Production-Scale Airport Boarding Model

Future development should focus primarily on richer datasets rather than replacing the movement engine. Boarding groups, seat assignments, walking speeds, cabin baggage, family groups, assisted passengers, aircraft types, multiple boarding doors and overhead locker behaviour can all be introduced while preserving the same local movement rules.

17. Reflection

This separation between computational engine and operational dataset is one of the principal lessons of the Algorithmic Domain Translation Series.

Concluding Remarks

This volume demonstrates that a boolean-based movement algorithm can be translated into a realistic airport boarding simulation while preserving its computational architecture. The airport scenario is only one interpretation. The same movement engine can form the basis of simulations for hospital patient flow, warehouse logistics, manufacturing systems and emergency evacuation.

The purpose of the Algorithmic Domain Translation Series is therefore broader than airport boarding: it is to reveal the shared computational structures that underpin many apparently different operational systems.

18. Appendices

Appendix A – Original Variable Mapping

This becomes a quick-reference table.

Original Algorithm	Airport Simulation	Purpose
people[]	boardingAisle[]	Aircraft aisle occupancy array
1	Passenger	Occupied aisle tile
0	Empty aisle tile	Free movement position
moveRight	Move toward rear seats	Boarding direction
moveLeft	Move toward front door	Reverse/deplaning scenario
recordPersonMoved[]	recordPassengerFinishedAt[]	Approximate identity tracking
hasPersonMove	hasPassengerMove	Detect movement
hasPersonPrevMove	hasPassengerPrevMove	Detect movement continuation
numberScansRow	numberBoardingTicks	Simulation phase counter

Appendix B – Airport Terminology

Term	Meaning
Aircraft door	Simulation start boundary
Boarding tick	One complete scan of the aisle
Aisle tile	One array index
Passenger	Anonymous occupancy value (1)
Empty tile	Array value 0
Boarding boundary	End of aisle model
Horizontal seating	Terminal transition outside the aisle model
Congestion	Passenger unable to move because next tile is occupied
Queue	Internal aircraft aisle queue only

Appendix C – Simulation Assumptions

The present simulation intentionally makes the following assumptions:

- Movement occurs one aisle tile per boarding tick.
- Only one-dimensional aisle movement is modelled.
- Seat entry occurs after leaving the aisle model.
- Passengers remain anonymous.
- All movement is deterministic.
- Aircraft-door entry marks simulation start.
- Final seating marks simulation end.
- Terminal operations are not simulated directly.

These assumptions deliberately preserve the intrinsic behaviour of the original algorithm while simplifying the surrounding operational environment.

19. Publication Edition Statement

Algorithmic Domain Translation Series
Volume 1 – Airport Boarding Simulation Publication Edition Version 1.0

This publication preserves the intrinsic structure of the original boolean movement algorithm while demonstrating how the same computational engine can be interpreted within a realistic operational domain. It is intended to serve as the reference template for future domain translation volumes.