Ticket Verification System – Systems Design & UML Case Study

Part I: Use Case Analysis

Case #	1
Step 1: primary actor and goal	Ticket scanner updates schedule
Step 2: main scenario (cannot be simpler, always ok, happily ends)	1. Records arrival and departure time of all trains. 2. Schedule arrival and departure times = new arrival and departure times.
Step 3: Failures	**Reason** a. Schedule arrival times don't match new arrival times. b. Schedule departure times don't match new departure times. **Detect at:** - When the scanner completes updating schedule records. **System response:** - Flag update failure and log exception for manual review
Step 4: alternative scenarios (from exceptions in the above)	1. Reattempt update, if times match then case ends. 2. If mismatch persists, end case with 'UpdateFailed' status.

Case #	2
Step 1: primary actor and goal	Ticket scanner allows travel on next available train if schedule changes
Step 2: main scenario (cannot be simpler, always ok, happily ends)	**Precondition:** Case 1 is successful (schedule is updated) 1. Detects next train's arrival time is not equal to prior arrival time. 2. Identifies train with earliest arrival time on same route. 3. Marks that train as next valid train for passenger travel.
Step 3: Failures	**Reason:** a. Schedule is outdated (case 1 exception) b. Next train exists but is outside service time (4:30-11:30) of current date. c. No next train available on same route **Detect at:** - When scanner is reassigning e-ticket to next marked train on updated schedule. **System response:** a. Reattempt case 1 if schedule is outdated b. Flag failure and log for manual review is above step fails.
Step 4: alternative scenarios (from exceptions in the above)	Reason a: Trigger Case 1, then retry case 2 Reason b: End case with status "OutOfServiceWindow", advise passenger to wait until next calendar date. Reason c: End case with status "NoNextTrainFound", prompt for alternative route.

Case #	3
Step 1: primary actor and goal	Ticket scanner confirms ticket matches with train
Step 2: main scenario (cannot be simpler, always ok, happily ends)	**<u>Precondition:</u>** Case 2 is successful (schedule is updated) 1. Reads the date and route of passenger's e-ticket. 2. Checks if ticket is one-way or round trip. 3. Confirms date and route of e-ticket matches with date and route of current train at arrival.
Step 3: Failures	**<u>Reason</u>** - E-ticket date or route does not match with current train. **<u>Detect at:</u>** - When passenger scans e-ticket at boarding entrance **<u>System response:</u>** - Reject e-ticket and display "InvalidTicket", prompt passenger to verify date and route.
Step 4: alternative scenarios (from exceptions in the above)	- End case with status "InvalidTicket".

Case #	4
Step 1: primary actor and goal	Ticket scanner enforces same-day travel within service window to prevent reuse outside valid time limits
Step 2: main scenario (cannot be simpler, always ok, happily ends)	**<u>Precondition:</u>** Case 3 succeeded (ticket matched current train) a. Scanner identifies ticket type (one-way or round trip). b. Confirms scan time is within service window (4:30am-11:30pm). c. If one-way: verifies ticket not previously used, then marks it as used. d. If round-trip: a. If first leg unused → mark outbound leg used. b. If outbound leg already used → check current scan date equals outbound scan date, then mark return leg used.
Step 3: Failures	**<u>Reason:</u>** a. Scan time is outside service window (4:30am-11:30pm) b. Reuse detected – one-way ticket already used or round-trip both legs already used. c. Round-trip second leg not same day as outbound. **<u>Detect at:</u>** - During time-window and usage-state checks immediately after the scan. **<u>System response:</u>** – Reject e-ticket and log reason code: OutOfServiceWindow / AlreadyUsed / NextDayReuse.
Step 4: alternative scenarios (from exceptions in the above)	– End case with the corresponding status (**OutOfServiceWindow**, **AlreadyUsed**, or **NextDayReuse**).

Case #	5
Step 1: primary actor and goal	Passenger scans e-ticket at station entry to board on scheduled train correspondent to ticket.
Step 2: main scenario (cannot be simpler, always ok, happily ends)	**Precondition:** Passenger possesses a valid e-ticket for current date. 1. Passenger presents or scans the e-ticket at the scanner device. 2. Ticket scanner reads ticket data (QR or NFC) and triggers verification sequence (Cases 1–4). 3. Scanner displays ticket status (Valid / Invalid). 4. If valid, passenger proceeds to train boarding.
Step 3: Failures	**Reason:** a. Scanner cannot read the ticket (damaged QR, poor connectivity, or NFC failure). b. Ticket rejected due to schedule, date, or service-window violations detected in linked cases. c. System timeout or network interruption prevents verification. **Detect at:** – When the passenger's ticket is scanned and the scanner cannot complete verification. **System response:** – Display message to passenger (e.g., "Scan Failed," "Invalid Ticket," or "Network Error"). – Log failure for follow-up and alert system operator if error persists.
Step 4: alternative scenarios (from exceptions in the above)	– End case with the corresponding status (**OutOfServiceWindow**, **AlreadyUsed**, or **NextDayReuse**).

Part II: Identifying Objects and Classes

Class 1:
Ticket

Unique ID: ticketID
Other attributes: type (ONE_WAY | ROUND_TRIP), routeCode, travelDate, usedOnceAt, outboundUsedAt, returnUsedAt
Autonomous action: validateAgainst(schedule: TrainSchedule): ValidationResult (checks route/date and one-way vs round-trip state consistent with Cases 3–4)
Relationships (for Part III):

- Ticket — **belongs to** → Passenger (1..* tickets per passenger)
- Ticket — **references** → TrainSchedule (1 ticket matches 1 schedule instance after Case 3)

Class 2:
Passenger

Unique ID: passengerID
Other attributes: name, contactEmail, accountID
Autonomous action: initiateScan(scanner: TicketScanner) *(your Case 5 trigger)*
Relationships:

- Passenger — **owns** → Ticket (1..*)

Class 3:
TrainSchedule

Unique ID: scheduleID
Other attributes: trainNumber, routeCode, stopId, arrivalTime, departureTime, status (ON_TIME | DELAYED | CANCELLED), updatedAt
Autonomous action: refreshFromSource() *(updates times/status so Cases 1–2 can consume "current" data)*
Relationships:

- TrainSchedule — **pertains to** → Train (many schedule rows per train)
- TicketScanner — **reads** → TrainSchedule

Class 4:
Train

Unique ID: trainID
Other attributes: routeCode, serviceHoursStart(04:30), serviceHoursEnd(23:30), capacity, active
Autonomous action: updateStatus(newStatus)
Relationships:

- Train — **has** → TrainSchedule (1..*)

Class 5:
TicketScanner

Unique ID: scannerID
Other attributes: locationId, scannerStatus, currentTimestamp
Autonomous action: updateSchedule() *(your Case 1 "dynamic schedule" maintenance; can trigger refresh)*
Relationships:

- TicketScanner — **validates** → Ticket
- TicketScanner — **reads** → TrainSchedule

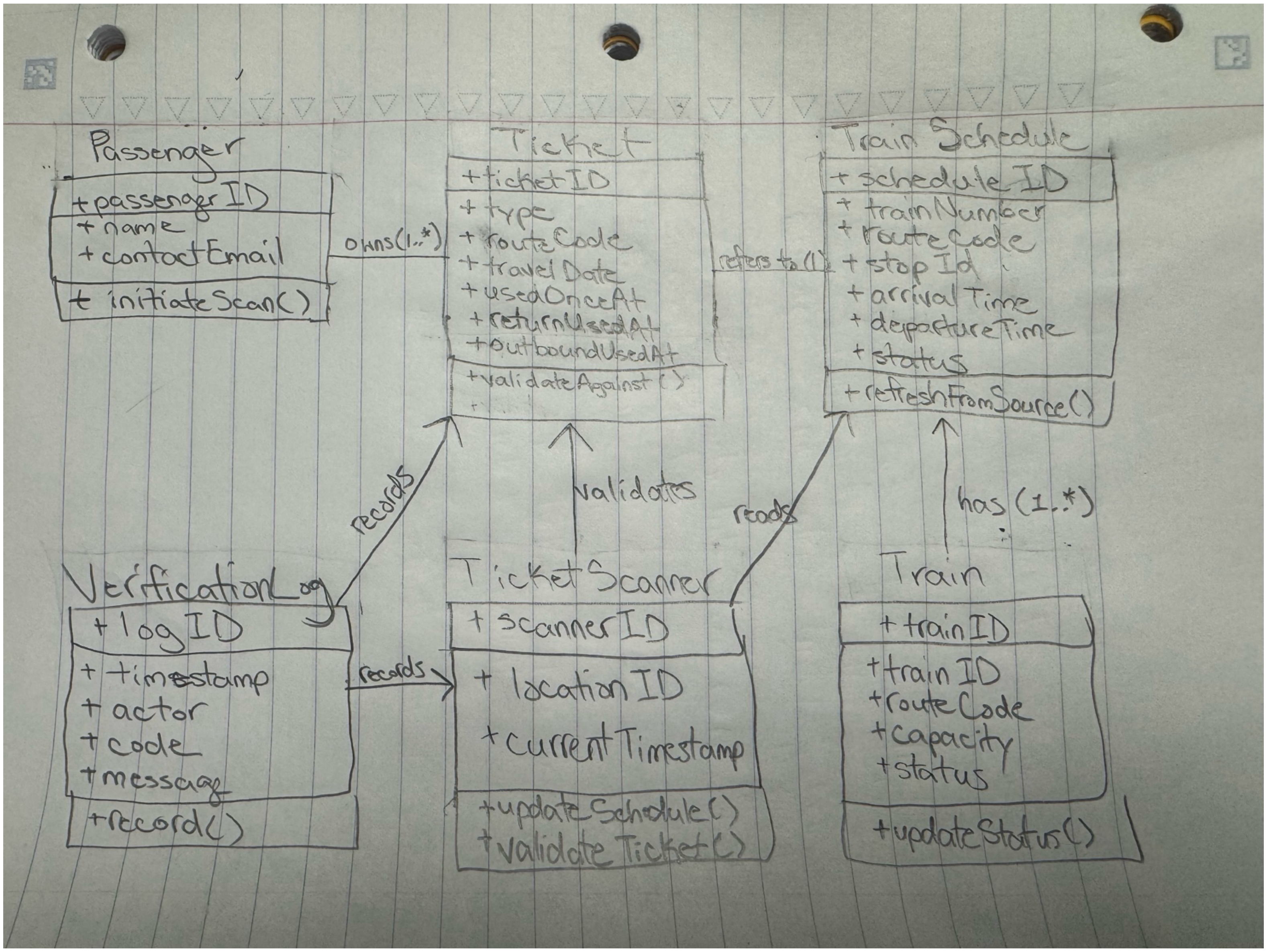
Class 6:
VerificationLog

Unique ID: logID
Other attributes: timestamp, actor (PASSENGER | SCANNER), ticketID, code (Valid | InvalidTicket | OutOfServiceWindow | NextDayReuse | AlreadyUsed | NoNextTrainFound | UpdateFailed), message
Autonomous action: record(event) *(persists each success/failure from Steps 3–4 across your cases)*
Relationships:

- VerificationLog — **records** → Ticket (0..*)
- VerificationLog — **records** → TicketScanner (0..*)

Part III: UML

The diagram below represents the finalized class relationships derived from the use cases above, emphasizing responsibility separation, validation flow, and logging.


Passenger
+passengerID
+name
+contactEmail
+ initiateScan()
owns(1..*)
Ticket
+ticketID
+type
+routeCode
+travelDate
+usedOnceAt
+returnUsedAt
+outboundUsedAt
+validateAgainst()
+
refers to (1)
Train Schedule
+scheduleID
+trainNumber
+routeCode
+stop Id
+arrivalTime
+departureTime
+status
+refreshFromSource()
records
validates
reads
has (1..*)
VerificationLog
+logID
+timestamp
+actor
+code
+message
+record()
records
TicketScanner
+scannerID
+locationID
+currentTimestamp
+updateSchedule()
+validateTicket()
Train
+trainID
+trainID
+routeCode
+capacity
+status
+updateStatus()