# Overview

This Detailed Level Design (DLD) documents the Release 2 technical design for the Railway Ticket Sales system. It defines a three-layer architecture (UI, Logic, Data), the UML class model (classes, attributes, methods, inheritance, interfaces), responsibilities for each class and method, enumerations and constants, how data flows through the system, traceability to the SRS, source file mapping, and a short revision history. Release 2 scope: structural refactor, separation of concerns, in-memory repositories, no business logic in main, no direct time/console use in domain classes, and replacement of hard-coded values with enums/constants. Validation, exception policies, file persistence, and full behavior diagrams are deferred to Release 3.

# System Overview / Architectural Context

Architecture follows a three‑layer structure with one‑way dependencies (UI 🡪 Logic 🡪 Data):

* Presentation Layer (UI): ConsoleUI – responsible only for user interaction, input parsing, and textual output; it translates user actions into service calls and displays results or errors, it does not implement business rules or access storage directly.
* Logic Layer (Business): TicketService, IClock, Employee abstractions, utility helpers – contains domain rules and flows (search 🡪 reserve 🡪 purchase 🡪 return), refund and profit calculations, transaction registry, and orchestration of repository updates; depends on abstractions (IClock, repository interfaces) rather than concrete storage.
* Data Layer (Storage): TicketRepository, PassengerRepository, TrainRepository (stub) – in‑memory storage and query/update primitives (add, find, getById, updateStatus, adjustBalance); persistence is intentionally deferred to a separate Persistence module (CSV/DB) in a later release.

Design principles: Separation of Concerns, Single Responsibility, Dependency Inversion (low coupling via small interfaces). UI never mutates repository state directly; Data layer never performs presentation; business logic is isolated for testability and deterministic unit tests. Main.cpp only wires components, seeds sample data, and hands control to ConsoleUI.

# UML Class Diagram (Technical Design)

…

# Class Specifications

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class** | **Type** | **Description** | **Attributes** | **Methods** |
| ConsoleUI | UI | Console interaction and formatting; prompts and renders results | mTicketRepo; mPassengerRepo; mService | displayMenu(); run(); listTickets(); searchAvailable(); registerPassenger(); purchaseFlow(); returnFlow(); showRegistryAndProfit() |
| TicketService | Service | Coordinates search, reserve, purchase, return flows; refund and profit logic; transaction registry | mTicketRepo; mPassengerRepo; mRegistry; mClock | TicketService(...); searchAvailable(destination,date,coach); completePurchase(passport,ticketId,outMessage); completeReturn(passport,ticketId,outMessage); calculateRefund(cost,daysBefore); getRegistry(); dailyProfit() |
| IExchangeOperation (planned) | Interface | (Planned abstraction for exchange-style operations; not required by current code) | - | processExchange(amount,from,to,rate) -> double |
| Ticket | Domain model | Ticket identity, booking state and CSV serialization | mId; mDate; mCost; mDestination; mCoachType; mStatus | Ticket(...); getId(); getDate(); getCost(); getDestination(); getCoachType(); getStatus(); setStatus(Status); showDetails(); toCSV() |
| Types / Enums | Utility | Enums and human-readable converters used across UI and services | enum Status; enum Coach | StatusToString(Status); CoachToString(Coach) |
| Transaction | DTO | Registry record for purchases and returns | ticketId; operation; timestamp; amount | (aggregate data only; no methods) |
| IClock | Interface | Abstract time source to decouple date logic and enable deterministic tests | - | nowISO(); daysBetween(dateIso) |
| SimpleClock (factory) | Concrete | Default IClock implementation and factory makeSimpleClock() | - | nowISO(); daysBetween(dateIso); makeSimpleClock() |
| TicketRepository | Repository | In-memory storage and query/update primitives for tickets | mTickets | addTicket(const Ticket&); listAll() const; findAvailable(destination,date,coachFilter) -> vector<int>; getById(id) -> Ticket*; updateStatusById(id,newStatus) -> bool; getByIndex(idx) -> Ticket* |
| PassengerRepository | Repository | In-memory passenger records: balances and purchased tickets | mPassengers (PassengerRecord: passport,balance,purchasedTickets) | addPassenger(passport,balance); getPassenger(passport) -> PassengerRecord\*; adjustBalance(passport,delta) -> bool; getBalance(passport) const; addPurchasedTicket(passport,ticketId); getPurchasedTickets(passport) const |
| TrainRepository | Repository (stub) | Placeholder for train/schedule data | - | addTrain(trainId) |
| Employee | Demo role | Simple inheritance demo for wiring and roles | mName | Employee(name); virtual role() const = 0; name() const |
| Clerk | Demo role | Concrete employee role used in startup wiring | (inherited) | Clerk(name); role() const override |
| Administrator | Demo role | Concrete employee role used in startup wiring | (inherited) | Administrator(name); role() const override |
| Util | Utility | Small helpers for timestamps and monetary rounding | - | nowTimestamp(); round2(float) |

# Interfaces and Abstractions

|  |  |  |  |
| --- | --- | --- | --- |
| **Interface** | **Purpose** | **Key Methods** | **Planned For (Release)** |
| IClock | Decouple system time for deterministic logic and testing | nowISO(); daysBetween(dateIso) | 2 |
| TicketService (service boundary) | Facade for domain operations; coordinates repositories, time source, and business rules | searchAvailable(dest,date,coach); completePurchase(passport,ticketId,outMsg); completeReturn(passport,ticketId,outMsg); calculateRefund(cost,daysBefore) | 2 |
| Repository abstractions (IRepository<T> planned) | Abstract data access to allow swapping in persistent stores and fakes for tests | get(id); save(T); list() -> std::vector<T> | 3 |
| IExchangeOperation (planned) | Enable polymorphic exchange/processing behaviors (fees, channels) | processExchange(amount,from,to,rate) -> double | 3 |
| ReportService / IReportable (planned) | Standardize report generation from domain state and allow interchangeable report generators | buildDailyReport(balances,profit); (IReportable) generate() -> Report | 2 (ReportService implemented); 3 (IReportable) |

# Function Responsibilities

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Class** | **Method** | **Purpose** | **Input** | **Output** | **Notes** |
| TicketService | searchAvailable | Query repository for matching available tickets | destination (string), date (ISO string), coachFilter (optional Coach) | vector<int> (ticket IDs or indices) | Read-only; no state mutation; filters Status::Available. |
| TicketService | completePurchase | Validate passenger and balance, charge, mark ticket sold, record transaction | passport (string), ticketId (int), outMessage (string& for user) | bool (success) | Adjusts PassengerRepository balance, updates Ticket status, appends Transaction to registry; returns descriptive outMessage on failure. |
| TicketService | completeReturn | Validate ownership and refund window, compute refund, credit passenger, mark ticket available, record transaction | passport (string), ticketId (int), outMessage (string& for user) | bool (success) | Uses IClock.daysBetween to determine refund bracket; calls calculateRefund; enforces service invariants. |
| TicketService | calculateRefund | Deterministic refund computation and rounding | ticketCost (float), daysBefore (int) | float (refund amount) | Pure function; uses Util::round2 for two-decimal monetary rounding. |
| TicketService | getRegistry / dailyProfit | Expose transaction history and aggregate profit | - | vector<Transaction> / float | Read-only views of session registry used by reporting and tests. |
| TicketRepository | addTicket / listAll | Store and enumerate tickets in memory | Ticket | void / vector<Ticket> | In-memory only in R2; no persistence; append semantics. |
| TicketRepository | findAvailable | Locate available tickets matching criteria | destination, date, optional coach | vector<int> (IDs/indices) | Returns candidates for UI selection. |
| TicketRepository | getById / updateStatusById | Lookup and mutate ticket state | id (int) / (id, Status) | Ticket\* / bool | Minimal bounds checking; business validation kept in TicketService. |
| PassengerRepository | addPassenger / getPassenger | Manage passenger records and balances | passport (string), balance (float) | void / PassengerRecord\* | In-memory store for balances and purchased ticket list. |
| PassengerRepository | adjustBalance / addPurchasedTicket | Change balance and record ownership | passport, delta / passport, ticketId | bool / void | adjustBalance enforces non-negative balance semantics in R2 basic guard. |
| IClock (SimpleClock) | nowISO / daysBetween | Provide current timestamp and date arithmetic | — / dateIso (string) | string (ISO) / int (days) | Injected into TicketService to make time-dependent rules testable and deterministic. |
| ConsoleUI | run / purchaseFlow / returnFlow / showResult | Present menu, collect input, and display results | user input via console | printed output; return codes | Performs only syntactic validation; delegates business rules to TicketService; uses repository methods for read-only listing. |
| Util | nowTimestamp / round2 | Provide canonical timestamp and monetary rounding | - / float value | string / float | Shared helpers used by service and transaction timestamping. |
| ReportService | buildDailyReport | Aggregate balances and profit into Report DTO | balances (map<Currency,double>), profit (double) | Report | Pure aggregation; formatting delegated to ConsoleUI. Implemented as concrete service in R2. |
| Employee / Clerk / Administrator | constructors / role / name | Lightweight role objects for wiring and demo | name (string) | role string / name string | Demo-only; no authorization logic; used in startup wiring and examples. |

# Operational flow

*Step 1 – User interaction (Presentation)*

* ConsoleUI displays the main menu and prompts the user for an action (list, search, purchase, return, show registry).
* ConsoleUI performs only basic syntactic checks (non-empty inputs, numeric parsing) and translates raw input into typed parameters (passport, ticketId, date, coach, amount).
* ConsoleUI calls the service boundary (TicketService) with the validated parameters.

*Step 2 – Service delegation (Logic)*

* ConsoleUI invokes TicketService methods (searchAvailable, completePurchase, completeReturn, getRegistry) to execute user intent.
* TicketService is the orchestrator for domain flows; it does not perform console I/O.

*Step 3 – Lookups and preconditions*

* TicketService queries TicketRepository for candidate tickets (findAvailable / getById) and PassengerRepository for passenger records (getPassenger).
* TicketService verifies structural preconditions (ticket exists, status is Available/Reserved/Sold, passenger exists).

*Step 4 – Time-dependent checks*

* When a flow requires date logic (refund eligibility, refund amount), TicketService calls IClock.daysBetween or nowISO to compute days-to-travel and timestamp transactions.
* IClock is injected, enabling deterministic behavior in tests.

*Step 5 – Business decision and computation*

* For purchases: TicketService computes costs, checks passenger balance, and determines side effects (charge amount).
* For returns: TicketService computes refund via calculateRefund(ticketCost, daysBefore) and rounds with Util::round2.
* All business rules (ownership checks, refund bracket, penalty) are encoded in TicketService.

*Step 6 – State mutation (Data)*

* TicketService requests repository operations to persist state changes: PassengerRepository.adjustBalance, PassengerRepository.addPurchasedTicket, TicketRepository.updateStatusById.
* Repositories implement in‑memory updates and basic guards; they do not perform business validation beyond structural checks.

*Step 7 – Registry, profit and reporting*

* TicketService appends a Transaction record to its in‑memory registry for each completed operation.
* TicketService exposes dailyProfit() and getRegistry() for reporting.
* ReportService (concrete) consumes the aggregated balances and profit to produce a Report DTO (buildDailyReport). Formatting and presentation of the report is done by ConsoleUI.

*Step 8 – Presentation of outcome*

* ConsoleUI receives results and outMessages from TicketService and prints:
  + confirmation or error messages
  + updated ticket lists or balances
  + the daily report (Report.toString) when requested
* ConsoleUI remains presentation-only and never mutates domain state directly.

**Future work (Release 3): introduce durable persistence (IRepository<T> implementations and FileService/DB), comprehensive validation and exception handling, configurable alerts and thresholds, a polymorphic IExchangeOperation for fee/channel strategies, formal sequence/state diagrams, and CI-backed integration tests to verify end-to-end persistence and error recovery.**

## Principle Summary

|  |  |
| --- | --- |
| **Concept / Principle** | **Realization in Release 2** |
| Abstraction & Encapsulation | Business rules and time logic are hidden behind small interfaces (IClock) and service boundaries (TicketService), keeping implementation details inside src/ and exposing narrow contracts in include/. |
| Separation of Concerns | Clear one‑way layering: ConsoleUI (Presentation) → TicketService (Logic) → Repositories (Data); UI handles I/O only, logic handles rules, data handles storage. |
| Polymorphism & Inheritance | Lightweight use of polymorphism for test seams (IClock) and role examples (Employee/Clerk/Administrator); future expansion point for strategy objects (IExchangeOperation). |
| Dependency Inversion (DIP) | High‑level modules depend on abstractions passed by constructor/reference (IClock, repository interfaces by concrete types now, planned generic IRepository<T> later) rather than hardcoded globals. |
| Single Responsibility (SRP) | Each class has a focused duty: Ticket holds state/serialization, TicketService orchestrates flows, repositories manage storage, ConsoleUI formats and presents. |

## Operation Flow (Release 2)

**A diagram of a service

AI-generated content may be incorrect.**