SRS-DLD

**S/W Detailed Level Design**

|  |  |  |  |
| --- | --- | --- | --- |
| **Project Name** |  | | |
| **Block Name** |  | | |
| **Author** |  | **Approver** |  |
| **Team** |  | | |

Contents

[1. Overview 4](#_Toc210821152)

[2. System Overview / Architectural Context 5](#_Toc210821153)

[3. UML Class Diagram (Technical Design) 6](#_Toc210821154)

[4. Class Specifications 6](#_Toc210821155)

[5. Interfaces and Abstractions 6](#_Toc210821156)

[6. Function Responsibilities 6](#_Toc210821157)

[7. Operation Flow 7](#_Toc210821158)

[8. Enumerations & Constants 7](#_Toc210821159)

[9. Validation Rules & Future Work 7](#_Toc210821160)

[10. Traceability Matrix 7](#_Toc210821161)

[11. Code Structure and File Mapping 7](#_Toc210821162)

[12. Revision History 8](#_Toc210821163)

* Revision History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Date | Revised contents | Author | Approver |
| **1.0** | 27.10.2025 | **Chapters 1-4** | Günther Miklas |  |
| **1.1** | 27.10.2025 | Chapters 5-11 | Klaus Jesper Zaletajev |  |

* Terms and Abbreviations

|  |  |
| --- | --- |
| **Term** | **Description** |
| **UI (User Interface)** | The component layer responsible for interaction between the user and the system. In this project, implemented through the ConsoleUI class. |
| **CSV (Comma-Separated Values)** | A simple text format used to store tabular data such as reserves, rates, and people in the DataStore class. |
| **ID (Identifier)** | A unique integer assigned to entities like employees, clients, and receipts to distinguish records across files and sessions. |
| ExchangeOffice | The core business logic class managing currency exchanges, reserves, profits, and reports. |
| Reserve | Represents the balance of currencies available for exchange, enforcing nonnegative constraints. |
| RateTable | Data structure maintaining conversion rates between currencies, including base-currency fallbacks. |
| Receipt | A transaction record issued after an exchange, containing details such as client, cashier, source amount, payouts, and profit. |
| DailyReport | The end-of-day summary generated by a manager, containing reserves, thresholds, transactions, and total profit. |
| Commission | The fee percentage applied to each transaction, representing the system’s profit margin (e.g., 3% in this implementation). |
| Critical Minimum | The minimum balance level for each currency reserve, below which a warning is triggered to prevent liquidity issues. |

* References

1. SW Requirements Specification

# Overview

This document describes a terminal-based Currency Exchange Management System. The system supports day-to-day exchange operations (cashier-handled client transactions), management controls (rates, reserves, end-of-day reporting), and simple file-based persistence (CSV/log/text reports).

**Scope and objectives**

* Execute currency exchanges with commission.
* Maintain reserve balances per currency with critical thresholds.
* Manage exchange rates.
* Produce end-of-day reports and append transaction logs.
* Persist rates, reserves, people, thresholds, and reports to files.

**Context**

* Runs as a console application (main.cpp) invoking ConsoleUI.
* No network/database; uses filesystem (CSV/log/text) for persistence.
* Business logic encapsulated in ExchangeOffice.
* Errors are signaled via domain exceptions

**Stakeholders & Roles**

* Client: Requests currency exchange.
* Cashier: Executes client exchanges and prints receipts.
* Manager: Adjusts rates/thresholds, inspects reserves, generates reports, resets daily cycle.
* Operations: Reads logs/reports, audits data.
* System: Filesystem for persistence; terminal for interaction.

# System Overview / Architectural Context

**Design**

* Clear separation of concerns:
  + UI Layer: Console I/O, input validation and flows.
  + Domain/Logic Layer: Currencies, requests, exchange rules, receipts, reporting.
  + Persistence Layer: File I/O for rates/reserves/people/thresholds/transactions/reports.

**Layers:**

**Presentation Layer**

* ConsoleUI (menu flows for Cashier/Manager)

**Logic Layer**

* ExchangeOffice, RateTable, Reserve, Receipt, DailyReport, BonusPolicy/PercentageBonusPolicy, domain models (ExchangeRequest, ExchangePortion, PayoutDetail)

**Data Layer**

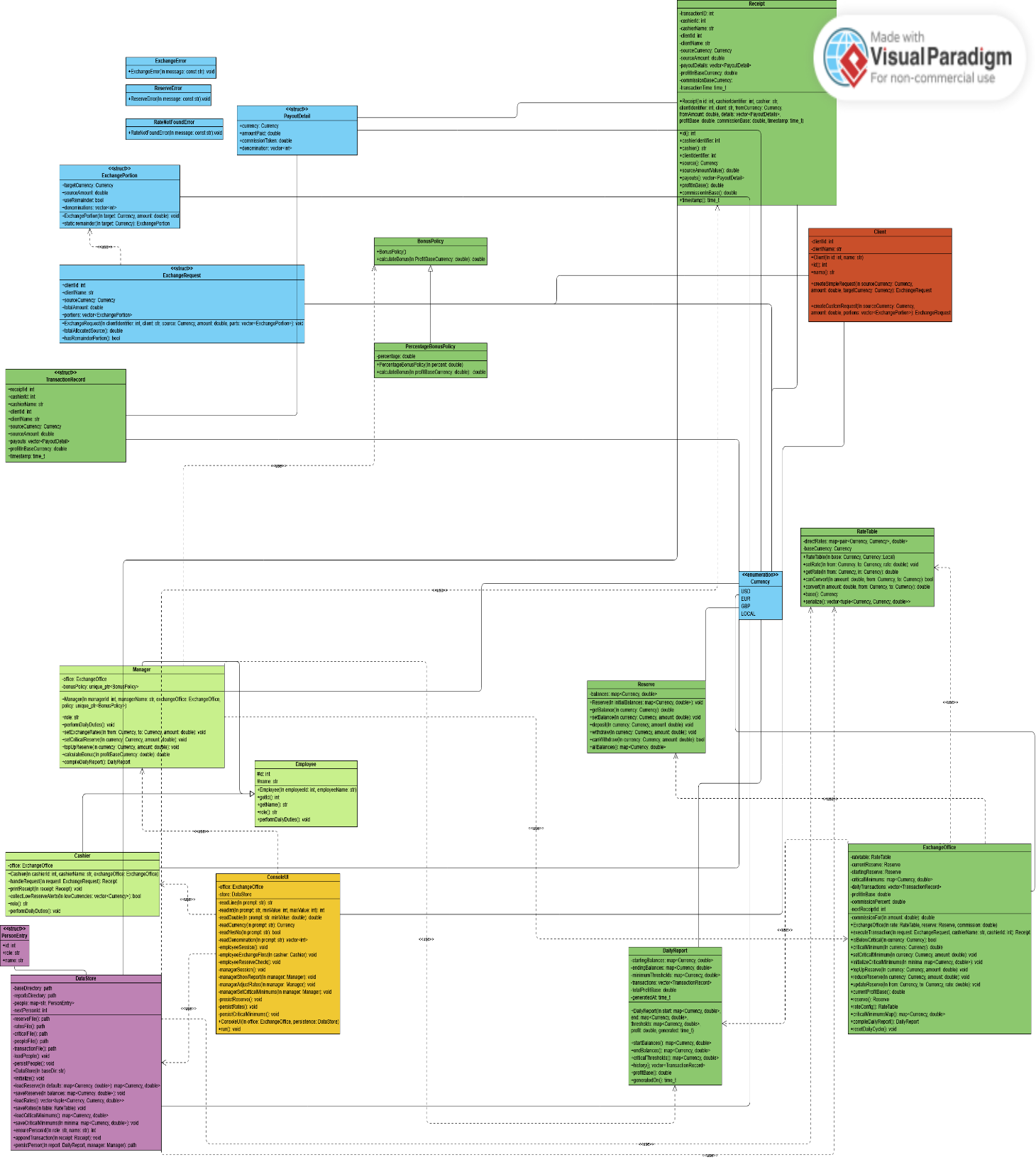
* DataStorage (+ PersonEntry) for CSV/log/text files

A screenshot of a computer

AI-generated content may be incorrect.

# UML Class Diagram (Technical Design)

Fully visible here: <https://online.visual-paradigm.com/share.jsp?id=343332323535372d31#diagram:workspace=gssyroho&proj=0&id=1>



# Class Specifications

**Core Utilities (utils.\*)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class** | **Type** | **Description** | **Attributes** | **Methods** |
| Currency | Enum class | Supported currency codes. | USD, EUR, GBP, LOCAL | --- |
| to\_string(Currency) | function | Stringify currency enum. | --- | std::string to\_string(Currency) |
| currency\_from\_string | function | Parse currency symbol (case-insensitive). | --- | Currency currency\_from\_string(const std::string&) |
| ExchangePortion | struct | Portion of a request allocated to a target currency. | Currency targetCurrency; double sourceAmount; bool useRemainder; std::vector<int> denominations; | ExchangePortion(Currency,double),  static ExchangePortion remainder(Currency) |
| ExchangeRequest | struct | A client’s exchange intent. | int clientId; std::string clientName; Currency sourceCurrency; double totalAmount; std::vector<ExchangePortion> portions; | ExchangeRequest(int,std::string,Currency,double,std::vector<ExchangePortion>),  double totalAllocatedSource() const, bool hasRemainderPortion() const |

**Exchange Management (exchange\_manager.\*)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class** | **Type** | **Description** | **Attributes** | **Methods** |
| Reserve | class | Tracks per-currency reserves with safety checks. | std::map<Currency,double> balances | Reserve(), explicit Reserve(map), double getBalance(Currency) const, void setBalance(Currency,double), void deposit(Currency,double), void withdraw(Currency,double), bool canWithdraw(Currency,double) const, const map& allBalances() const |
| RateTable | class | Manages direct rates and conversion via base currency. | map<pair<Currency,Currency>,double> directRates; Currency baseCurrency; | explicit RateTable(Currency base=LOCAL), void setRate(Currency, Currency, double), double getRate(Currency,Currency) const, bool canConvert(Currency,Currency) const, double convert(double, Currency, Currency) const, Currency base() const, vector<tuple<Currency,Currency,double>> serialize() const |
| TransactionRecord | struct | Immutable snapshot stored in report history. | int receiptId,cashierId,clientId; std::string cashierName,clientName; Currency sourceCurrency; double sourceAmount; vector<PayoutDetail> payouts; double profitInBaseCurrency; time\_t timestamp; | --- |
| Receipt | class | Client-visible result of a transaction. | IDs, names, Currency sourceCurrency; double sourceAmount; vector<PayoutDetail> payoutDetails; double profitBaseCurrency; double commissionBaseCurrency; time\_t transactionTime; | Ctor with all fields; accessors: id(), cashierIdentifier(), cashier(), clientIdentifier(), client(), source(), sourceAmountValue(), payouts(), profitInBase(), commissionInBase(), timestamp() |
| DailyReport | class | End-of-day summary (balances, thresholds, transactions, profit). | map start,end,thresholds; vector<TransactionRecord> transactions; double totalProfitBase; time\_t generatedAt; | Ctor; accessors: startBalances(), endBalances(), criticalThresholds(), history(), profitInBase(), generatedOn() |
| BonusPolicy | interface | Strategy for computing bonuses. | --- | virtual double calculateBonus(double) const = 0 |
| PercentageBonusPolicy | class | Percentage-based bonus strategy. | double percentage; | explicit PercentageBonusPolicy(double), double calculateBonus(double) const |

**Employees & Client (employee.\*, client.\*)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class** | **Type** | **Description** | **Attributes** | **Methods** |
| Employee | abstract class | Base type for system users with duties. | int id; std::string name; | Employee(int,std::string), virtual ~Employee(), int getId() const, const std::string& getName() const, virtual std::string role() const = 0, virtual void performDailyDuties() = 0 |
| Cashier | class | Executes exchanges and prints receipts; monitors critical reserves. | ExchangeOffice& office (+ inherited) | Cashier(int,std::string,ExchangeOffice&), Receipt handleRequest(const ExchangeRequest&), void printReceipt(const Receipt&) const, bool collectLowReserveAlerts(std::vector<Currency>&) const, overrides: role(), performDailyDuties() |
| Manager | class | Manages configuration, generates reports, computes bonuses. | ExchangeOffice& office; std::unique\_ptr<BonusPolicy> bonusPolicy; | Manager(int,std::string,ExchangeOffice&, std::unique\_ptr<BonusPolicy>=std::make\_unique<PercentageBonusPolicy>(0.05)), overrides: role(), performDailyDuties(), mgmt ops: setExchangeRate, setCriticalReserve, topUpReserve, double calculateBonus(double) const, DailyReport compileDailyReport() const |
| Client | class | Represents a customer initiating exchange. | int clientId; std::string clientName; | Client(int,std::string), int id() const, const std::string& name() const, ExchangeRequest createSimpleRequest(Currency,double,Currency) const, ExchangeRequest createCustomRequest(Currency,double,std::vector<ExchangePortion>) const |

**Persistence and UI (persistence.\*, console\_ui.\*, main.cpp)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class** | **Type** | **Description** | **Attributes** | **Methods** |
| PersonEntry | struct | Stored identity for people. | int id; std::string role; std::string name; | --- |
| DataStore | class | File-backed repositories for reserves, rates, people, thresholds, transactions, and reports directory. | Paths: baseDirectory, reportsDirectory; map<std::string,PersonEntry> people; int nextPersonId; | explicit DataStore(const std::string& baseDir="data"), void initialize(), **Reserves**: map loadReserve(const map& defaults) const, void saveReserve(const map&) const; **Rates**: vector<tuple<Currency,Currency,double>> loadRates() const, void saveRates(const RateTable&) const; **Criticals**: map loadCriticalMinimums() const, void saveCriticalMinimums(const map&) const; **People**: int ensurePersonId(const std::string& role, const std::string& name); **Audit**: void appendTransaction(const Receipt&) const, std::filesystem::path persistReport(const DailyReport&, const Manager&) const |
| ConsoleUI | class | CLI workflows for employees and managers; input validation and persistence hooks. | ExchangeOffice& office; DataStore& store; | ConsoleUI(ExchangeOffice&, DataStore&), void run(), (private helpers) readLine, readInt, readDouble, readCurrency, readYesNo, readDenominations; flows: employeeSession, employeeExchangeFlow(Cashier&), employeeReserveCheck() const, managerSession, managerShowReport(Manager&), managerAdjustRates(Manager&), managerSetCriticalReserve(Manager&); persistence: persistReserve() const, persistRates() const, persistCriticalMinimums() const |
| main.cpp | program | Entry point: validates args, boots DataStore, seeds defaults, wires objects, runs UI, persists on exit. |  |  |

# 5. Interfaces and Abstractions

|  |  |  |  |
| --- | --- | --- | --- |
| Interface | Purpose | Key Methods | Planned For (Release) |
| BonusPolicy | Enable polymorphic bonus calculation strategies. | calculateBonus(double profitBaseCurrency) double | 2 |
| Employee | Provide polymorphic behavior for all personnel types. | (employee)::performDailyDuties | 2 |
| IRepository | Abstract data access for persistence later. | get(id)->T; save(T)->void; list()->vector | 3 |
| IReportable | Uniform report generation. | generate() -> Report | 3 |

# 6. Function Responsibilities

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Class | Method | Purpose | Input | Output | Notes |
| ExchangeOffice | executeTransaction | Coordinate complete exchange transaction with multi-currency payout. | From currency, to currency, currency ammount | Receipt | Validates allocations, processes each portion, applies commission, updates reserves |
| ExchangeOffice | compileDailyReport | Generates end-of-day report | Stats | DailyReport | Aggregates all daily data into DTO |
| ExchangeOffice | resetDailyCycle | Clear daily transactions | none | void | Snapshots current reserves as new starting balances |
| RateTable | setRate | Store exchange rate | Curreny ID, Currency rate | void | Validates rate > 0. |
| RateTable | getRate | Gets the rate | rate of the desired currency | Rate | Returns direct rate or RateNotFoundError |
| RateTable | convert | Perform currency converts | Currency ID from, Currency ID To | void | Uses direct rate if available |
| Reserve | getBalance | Gets the balance of a user | User ID | Balance |  |
| Reserve | deposit | Add currency to reserve | Currency Id, Ammount of currency | void |  |
| Reserve | witdhdraw | Remove currency from reserve | Currency ID, Ammount of currency | void |  |
| ConsoleUI | run | Main application loop with menu navigation |  |  | Displays main menu, handles employee/manager login |
| ConsoleUI | managerShowReport | Generates and displays dailiy report | Manager ID | Dailiy report |  |

# 7. Operation Flow

**Step 1 – User interaction**

The user (cashier) starts the process by:

* Logging in with employee name
* Selecting "Perform exchange" from menu
* Entering client name
* Entering source currency (USD, EUR, GBP, or LOCAL)
* Entering total amount to exchange
* Choosing simple or split payout

It performs basic syntactic validation (non-empty, valid format, positive numbers)

**Step 2 –** **Delegation to Logic Layer**

ConsoleUI transfers control to:

Cashier::handleRequest(request)

ExchangeOffice::executeTransaction(request, cashierName, cashierId)

This marks the transition from Presentation layer to Logic layer.

**Step 3 – Validation and Initialization (Logic Layer)**

ExchangeOffice validates:

* Total allocated source ≤ request.totalAmount
* All amounts are positive
* Request has at most one remainder portion

Initializes tracking variables:

* remainingSource = request.totalAmount
* payoutDetails = [] (empty vector)
* profitBase = 0.0

**Step 4 – Process Each Portion (Logic Layer)**

Determine Source Amount:

* If portion.useRemainder == true: use remainingSource
* Else: use `portion.sourceAmount`
* Skip if amount < epsilon (very small/zero)

Rate Retrieval:

* ExchangeOffice requests rate from RateTable
* rate = rateTable.getRate(sourceCurrency, portion.targetCurrency)

RateProvider logic:

* If from == to: return 1.0
* Check direct rate map: if exists, return rate
* Try indirect via base: convert to base, then to target
* If no path: throw RateNotFoundError

Principle: Polymorphism - RateTable encapsulates conversion logic.

Conversion and Commission:

* convertedAmount = sourceAmount \* rate
* commission = convertedAmount \* commissionPercent (e.g., 3%)
* payoutAmount = convertedAmount - commission

Reserve Validation:

* Check if sufficient reserve:
* if (!reserve.canWithdraw(targetCurrency, convertedAmount)) throw ReserveError

Principle: Encapsulation - Reserve manages balance validation internally.

Reserve Update:

* reserve.withdraw(targetCurrency, convertedAmount) - Remove full converted amount
* reserve.deposit(targetCurrency, commission) - Return commission portion
* reserve.deposit(sourceCurrency, sourceAmount) - Add received source currency

Profit Tracking:

Convert commission to base currency for centralized accounting:

* commissionInBase = rateTable.convert(commission, targetCurrency, baseCurrency)
* profitBase += commissionInBase

Record Payout Detail:

* PayoutDetail {

currency: targetCurrency,

amountPaid: payoutAmount,

commissionTaken: commission,

denominations: portion.denominations

}

Add to `payoutDetails` vector.

Update Remaining:

* remainingSource -= sourceAmount`

Principle: Single Responsibility - Each step performs one clear task.

**Step 5 – Generate Receipt (Logic Layer)**

After processing all portions, ExchangeOffice creates immutable Receipt:

* Receipt {

id: nextReceiptId++ (auto-increment)

cashierId, cashierName

clientId, clientName

sourceCurrency, sourceAmount (total used)

payoutDetails (all portions)

profitBaseCurrency (accumulated)

commissionBaseCurrency (total commission)

transactionTime (current timestamp)

}

Record transaction:

* Create TransactionRecord and add to dailyTransactions vector.

**Step 6 – Display Result (UI Layer)**

ConsoleUI receives Receipt and formats output:

* Receipt #1 for client John Doe (ID 5) handled by Bob Smith (ID 1)

Source: USD amount 1000.00

-> GBP amount: 243.90 (denominations: 50, 20)

-> EUR amount: 368.00 (denominations: 100, 50)

-> LOCAL amount: 291.60 (denominations: 20, 10)

Profit (base): 29.16

Error Handling:

* If exception thrown (RateNotFoundError, ReserveError, ExchangeError):
* Catch at UI layer
* Display user-friendly message
* Transaction aborted (no reserves modified)

Principle: UI handles presentation, Logic handles errors semantically.

**Step 7 - Manager Daily Report Flow**

Report Request:

* Manager selects "Generate end-of-day report" from menu.

Compile Report (Logic Layer):

* Manager calls compileDailyReport() → ExchangeOffice creates DailyReport with:
* Starting balances (snapshot from cycle start)
* Ending balances (current reserves)
* Critical thresholds
* All transactions from dailyTransactions
* Total profit in base currency
* Current timestamp

Calculate Bonus (Logic Layer):

* Manager calls calculateBonus(report.profitInBase())
* Delegates to bonusPolicy->calculateBonus(profit)
* PercentageBonusPolicy: `return profit \* 0.05` (5%)

Display Report, consoleUI formats comprehensive report with:

* Generation timestamp
* Profit summary
* Ending reserves with critical threshold indicators
* Transaction list
* Calculated bonus

# 8. Enumerations & Constants

|  |  |  |
| --- | --- | --- |
| enum class Currency | {USD, EUR, GBP, Local} | Supported currencies for exchange operations |
| Const double commissionPercent | 0.03 | 3 % commission applied to all exchanges |
| Const double kEpsilon | 1e-8 | Floating-point comparison tolerance |
| Const double Default bonus rate | 0.05 | 5% bonus from dailiy profit |
|  |  |  |

# 9. Validation Rules & Future Work

|  |  |  |
| --- | --- | --- |
| Rule / Planned Feature | Description | Target Release |
| Denomination preferences (Coins) | Users can ask for coins also | Release 3 |
| Transaction rollback | Managers can initiate rollbacks for transactions | Release 3 |
|  |  |  |
|  |  |  |

# 10. Traceability Matrix

|  |  |  |
| --- | --- | --- |
| Requirement (SRS) | Class/ Method (DLD) | Notes |
| Convert Money between currencies | ExchangeOffice::executeTransaction | Suports direct conversions |
| Apply commission to exchanges | ExchangeOffice::commissionFor() | Applies a 3% commision |
| Present result | Cashier::printReceipt() |  |
| Reset daily cycle | ExchangeOffice::resetDailyCycle() | Clears transactions |
| Low-reserve alerts | Cashier::collectLowReserveAlerts() | Checks currencies against thresholds, displays console alerts |
|  |  |  |

# 11. Code Structure and File Mapping

| Class | File |
| --- | --- |
| ConsoleUI | Console\_ui.h / Console\_ui.cpp |
| Cashier | Employee.h /employee.cpp |
| Manager | Employee.h/ employee.cpp |
| Client | Client.h /client.cpp |
| Receipt | Exchange\_manager.h/ Exchange\_manager.cpp |
| DailyReport | Exchange\_manager.h/ Exchange\_manager.cpp |
| RateTable | Exchange\_manager.h/ Exchange\_manager.cpp |
| DataStore | Persistence.h/ Persistence.cpp |
| Reserve | Exchange\_manager.h/ Exchange\_manager.cpp |
| ProsentageBonusPolicy | Exchange\_manager.h/ Exchange\_manager.cpp |

# 12. Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Change Summary | Author |
| 28.10.2025 | 1.1 | Release 2 DLD example for Currency Exchange System. | Klaus Jesper Zaletajev |