

Project Documentation for Intuit Assignment 2: SA-001 Sales Analysis System

1. Introduction

The **Sales Analysis System** is a Java 21 application built to demonstrate functional programming proficiency, data transformation, and aggregation techniques using the Java Stream API.

The system processes structured CSV sales data and computes key business insights such as total revenue, product performance, customer value, and regional distribution. It is designed with testability, modularity, and extensibility in mind.

This document provides a comprehensive overview of:

- System architecture
- Data flow
- Component responsibilities
- Functional APIs
- Error handling
- Test strategy
- Performance considerations
- Future improvements

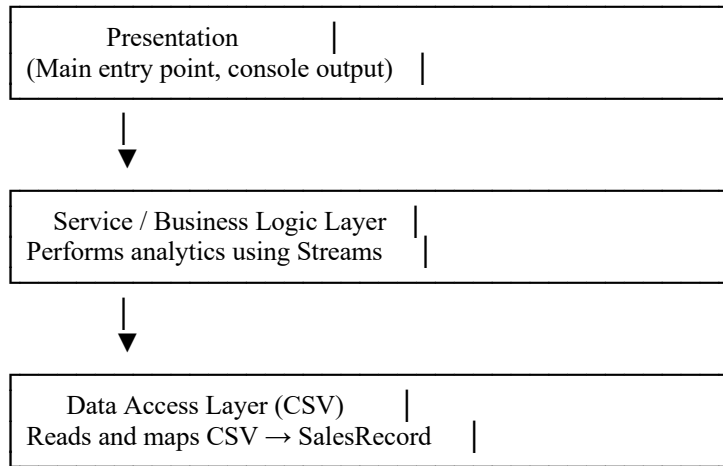
2. Objectives of the Project

The core objective of this assignment is to demonstrate mastery of:

- Functional programming in Java
- Stream operations (map, filter, reduce, collectors)
- Data aggregation techniques
- Lambda expressions
- Software design patterns
- Unit testing using JUnit 5
- Clean, maintainable, and modular code

3. System Architecture

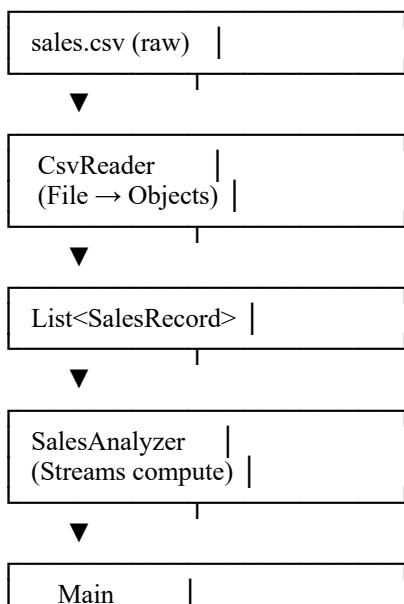
The architecture follows a **three-layer modular structure**:



3.1 Components

Module	Responsibility
CsvReader	Reads and parses CSV file into model objects
SalesRecord	Data model representing a row in CSV
SalesAnalyzer	Contains analytical methods using Java Streams
Main	Orchestrates data loading and prints results
SalesAnalyzerTest	Validates correctness of analytics via JUnit

4. Data Flow



(Console Output) |

5. Data Model

SalesRecord

Field	Type	Description
orderId	String	Unique order identifier
customer	String	Customer name
region	String	Region where sale occurred
product	String	Product type
quantity	int	Units sold
unitPrice	double	Price per unit
date	String	ISO date

Code Walkthrough

The flow of this project is simple, but each part has a clear responsibility. Everything begins with the **CsvReader**, which loads the CSV file and converts each line into a usable Java object. Instead of working with raw text, the reader turns every row into a **SalesRecord**. This class is just a POJO, a plain data holder meant to store information about a single sale. It doesn't perform any calculations; it simply keeps the data organized so the rest of the application can work with it easily.

- **CsvReader**
 - Opens the CSV file
 - Splits each line into individual fields
 - Converts text values into numbers where needed
 - Creates a **SalesRecord** object for every row
 - Returns a full list of all sales

Once the data is loaded, the real work happens inside the **SalesAnalyzer**. This class contains all the analytical logic and uses Java Streams to compute insights. Each method focuses on a different business question. For example, total revenue is found by multiplying quantity and unit price for each sale and summing everything. Grouping operations help determine how many orders came from each region or how much revenue each product generated. The analyzer also figures out the top customer, the most profitable product, the average order value, and more.

- **SalesAnalyzer**
 - Calculates total revenue

- Counts orders per region
- Groups revenue by product
- Finds highest-earning product
- Computes average order value
- Identifies top-spending customer
- Counts total units sold
- Splits orders into high-value vs. regular

All of this is tied together by the Main class. It loads the CSV, hands the data to the analyzer, and prints the results so the user can see the insights in a simple console format. The project finishes with a suite of JUnit tests. These tests check each analytical method by feeding small sample datasets and verifying that the output matches the expected result.

- **Main**
 - Loads the dataset
 - Runs all analysis methods
 - Prints results to the console
- **SalesAnalyzerTest**
 - Uses mock data to test each calculation
 - Prints expected vs actual values
 - Ensures all logic works correctly

6. Functional API Documentation (SalesAnalyzer)

6.1 totalRevenue(List) → double

Description: Computes the total revenue across all sales.

Logic:

- Maps each record to quantity * price
- Sums them using reduce

Edge Cases: Empty list → 0.0

6.2 groupOrdersByRegion(List) → Map<String, Long>

Description: Counts number of orders per region.

Logic: groupingBy(region, counting())

6.3 revenueByProduct(List) → Map<String, Double>

Description: Computes total revenue per product.

Logic:

- Group by product
- Map each record to revenue
- Reduce via summing

6.4 highestRevenueProduct(List) → Optional

Description: Determines which product generated the highest revenue.

Logic:

- Uses `collectingAndThen(maxBy)`
Edge Cases: Empty list → `Optional.empty`

6.5 meanOrderValue(List) → double

Description: Computes average revenue per order.

Logic: uses `DoubleSummaryStatistics`

6.6 mostValuableCustomer(List) → Optional

Description: Identifies customer with the highest total spend.

Logic:

- Group by customer
- `SummingDouble` on revenue

6.7 totalUnitsSold(List) → Map<String, Integer>

Counts total units sold for each product.

6.8 splitHighValueOrders(List, double) → Map<Boolean, List>

Partitions orders into:

- `true` → revenue \geq threshold
- `false` → below threshold

7. Error Handling Strategy

CsvReader

Handles:

- Missing file
- Malformed lines
- Non-numeric values
- Blank lines

Invalid rows are skipped gracefully without stopping execution.

SalesAnalyzer

Always returns a valid value:

- Empty lists → return 0, empty maps, or Optional.empty
 - No NullPointerExceptions: defensive coding through Streams

8. Testing Strategy

Unit tests validate:

- Functional programming constructs
- Stream correctness
- Aggregation accuracy
- Edge cases
- Robustness with anomalies

Tests implemented:

- Total revenue
- Region grouping
- Product grouping
- Highest revenue product
- Average order value
- Top customer
- Tie scenarios
- Negative values
- Large numeric values
- Empty dataset
- Order independence

9. Performance Considerations

Time Complexity

- All operations are linear: **$O(n)$**
- Grouping uses hashing: **$O(n)$**
- No nested loops

Space Complexity

- Uses small collector maps: **$O(k)$**
Where k = number of unique products/customers/regions

Scalability

- Java Streams handle large datasets efficiently
- Quick to parallelize using `.parallelStream()` if needed

10. Future Enhancements

- Replace manual CSV parsing with **SuperCSV/OpenCSV**
- Add JSON export of analytics
- Add REST API interface using Spring Boot
- Support monthly/quarterly/yearly rollups
- Add charts (bar, pie, timeline)
- Write custom collectors for educational purposes
- Use Java Records for immutable modeling
- Add persistence using SQLite or PostgreSQL

11. Glossary

Term	Meaning
Stream	Functional pipeline for data operations
Collector	Terminal operation that aggregates data
Optional	Wrapper representing presence or absence of value
Reducer	Functional pattern to fold elements into one
Partition	Split dataset into two buckets
Aggregation	Combining data into meaningful metrics